Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

GEORG-DANIEL SCHWARZ
MASTER THESIS

MIGRATING THE JVALUE ODS TO MICROSERVICES

Submitted on 20 March 2019

Supervisors:
Prof. Dr. Dirk Riehle, M.B.A.,
Andreas Bauer, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 20 March 2019

License

Erlangen, 20 March 2019

Abstract

„A world in which consuming open data is easy and safe.“

This is the vision of the JValue Open Data Service (ODS) that aims to bundle finding the right open data, using and combining it into an easy and intuitive process. The daily life of developers should be free from hardships, like writing data crawlers for different data formats in combination with different protocols, and instead, let them focus on their real problems.

As in every crowd-sourced application, the more users contributing and adding new data sources, the better. With these requirements, the need for a scalable software emerges. The microservice-based approach promises to achieve this, as well as making the project easier to understand and maintain for developers.

This thesis presents a microservice-based architecture draft for the ODS and a migration strategy from the current monolithic architecture towards it. The exemplary implementation of a selected microservice proofs the feasibility of such an architectural style. Furthermore, a discussion about the challenges and benefits of a distributed system and the experiences written down in this thesis shall reduce the risks and enable a complete migration to a microservices-based architecture in the future.

Contents

Acronyms

API Application Programming Interface

DDD Domain-Driven Design

JWT JSON Web Token

ODS Open Data Service

SOA Service-Oriented Architecture

REST Representational State Transfer

UI User Interface

URI Uniform Resource Identifier

1 Introduction

Nowadays we live in an age of data and information - the digital age. In the last few decades, many inventions like the internet led to the omnipresence of technology in our daily lives. Nearly everyone possesses a smartphone and almost every device we use incorporates sensors. With this digital revolution we were experiencing and are still taking part in, the amount of data collected per person is increasing rapidly. [RGR17]

Without processing this data barely provides value in most cases. The steps to derive benefit from the data involve automated routines because of its huge extent. This leads to one of the bigger challenges right now: how to deal with these amounts of data.

A significant share of this data is accessible by everyone. If anyone can use this data without restrictions, it is called Open Data. [Ope14]

One of the big challenges in the Open Data domain is the variety of data representations which different data sources use. If an application requires data of different sources, usually the developers need to spend additional effort for extracting and transforming data.

The JValue Open Data Service (ODS)[1] promises to provide the solution to this problem. Users can configure the system to extract data of an Open Data source and transform it into suitable representations. The transformed data is accessible for the user in a uniformed RESTful API. This implies for application developers, they don't have to deal with different data sources anymore after configuring the ODS and can access the data in a standardized format. The vision of the ODS also includes that the amount of connected data sources increases over time by crowd-sourcing. Rephrasing, users can benefit from already accessed data sources.

Over time, some complexity found its way into the ODS project. The main sources of complexity are on the one hand the ability to handle various data formats to support as many types of data sources as possible. On the other hand,

[1] GitHub project: `https://github.com/jvalue/open-data-service`

very individual data transformations have to be possible to provide various uses for the extracted data.

An adaption of the architecture promises to reduce this complexity and make the system easier to understand. Lower coupling and higher cohesion are the goals that promise better maintainability in this context. [HM95, p. 1]

The future use of the ODS as a crowd-sourced application influences this matter, so the architecture should enable suitable scaling capabilities in the long term. One solution for this is the introduction of a microservice-based architecture as section 2.5 shows.

This thesis defines a suitable microservice-based architecture with the primary goal to make the whole system easier to understand. Therefore, components in the ODS are identified that can be extracted into microservices. A process is defined on how to migrate a monolithic application to a microservice-based application. Risks are evaluated and additionally required technologies are discussed. The migration is exemplarily conducted in order to show such an approach is feasible for the ODS. Finally, there is an evaluation that examines if the described goals were met.

The thesis is structured in the following way:

Chapter 2 (Fundamentals: Microservices) introduces all fundamentals regarding microservices required for the understanding of this thesis.

Chapter 3 (Thesis Requirements) explains the detailed requirements for this thesis.

Chapter 4 (JValue Open Data Service) presents the vision of the ODS, analyzes the current state of the project and proposes a microservice-based architecture draft.

Chapter 5 (Migration of the Open Data Service) defines a process on how to migrate a monolithic application to a microservice-based architecture. Commonly in such architectures found technologies are presented in order to enable a seamless migration.

Chapter 6 (Implementation) shows that the proposed architecture and the migration process are both feasible by an example implementation. Some additional pitfalls regarding the consistency of the system are discussed.

Chapter 7 (Evaluation) recaps the accomplishments of the thesis and discusses whether the requirements were met.

2 Fundamentals: Microservices

2.1 Definition

Microservice-based architectures have no standardized definition, but there is some common sense about this architectural style. Martin Fowler defines it like this:

> „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.“ [LF14]

This definition gives a good impression on what microservices might be. The first important point is the modularization of a large software system into smaller services that communicate with each other. Those are independent of each other in the aspects of functionality, deployment and used technology. Typically, the communication happens over a network, hence this style of architecture describes a distributed system.

Eberhard Wolff explicitly emphasizes that these self-contained microservices each have their own data storage. This implies that it is a bad practice to share data between services on the database level. [Wol16, chap. 1.1]

Summarizing, these definitions cover the most important aspects regarding microservices. In order to emphasize some of these points, a comparison to a monolithic and a service-oriented architecture is drawn in the following two sections.

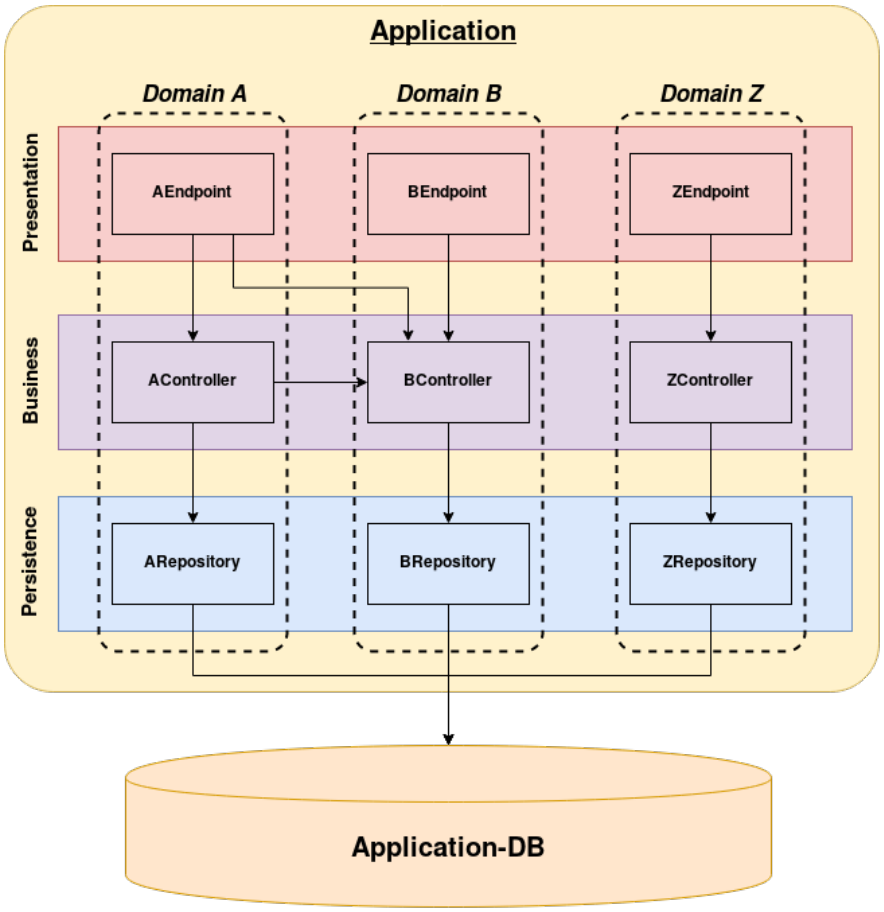


Figure 2.1: Layers in a Monolith according to [Sha17]

2.1.1 Comparison to the Monolith

A monolithic application typically contains all the functionality required by the system. A popular pattern is a layered approach which divides the system horizontally into multiple tiers - for example, one layer as an interface to the outside world, another one that contains the business logic and the third one interfaces with the data store. Often the implementation is also divided vertically into different domains in order to keep the coupling of components to a minimum. If domains are connected, method calls are used to interact with other tiers of the same level or of the ones below. All data may be stored in the end in the same database, using mechanisms like foreign keys if a connection between domains exists. Figure 2.1 shows this kind of architectural approach.

Figure 2.2 shows a typical application of the microservice-based architectural style. The service boundaries are aligned with the domains, each service may be implemented in layers again. Every service has its own data store as the definition

of section 2.1 states.

Compared to the monolith, pure method calls between domains are not possible. The communication has to be implemented over the interface of the services to the outside world. Technologies like HTTP calls are often used for synchronous communication that requires a response, asynchronous communication can be implemented by a message queue or also by HTTP calls for example. [LF14]

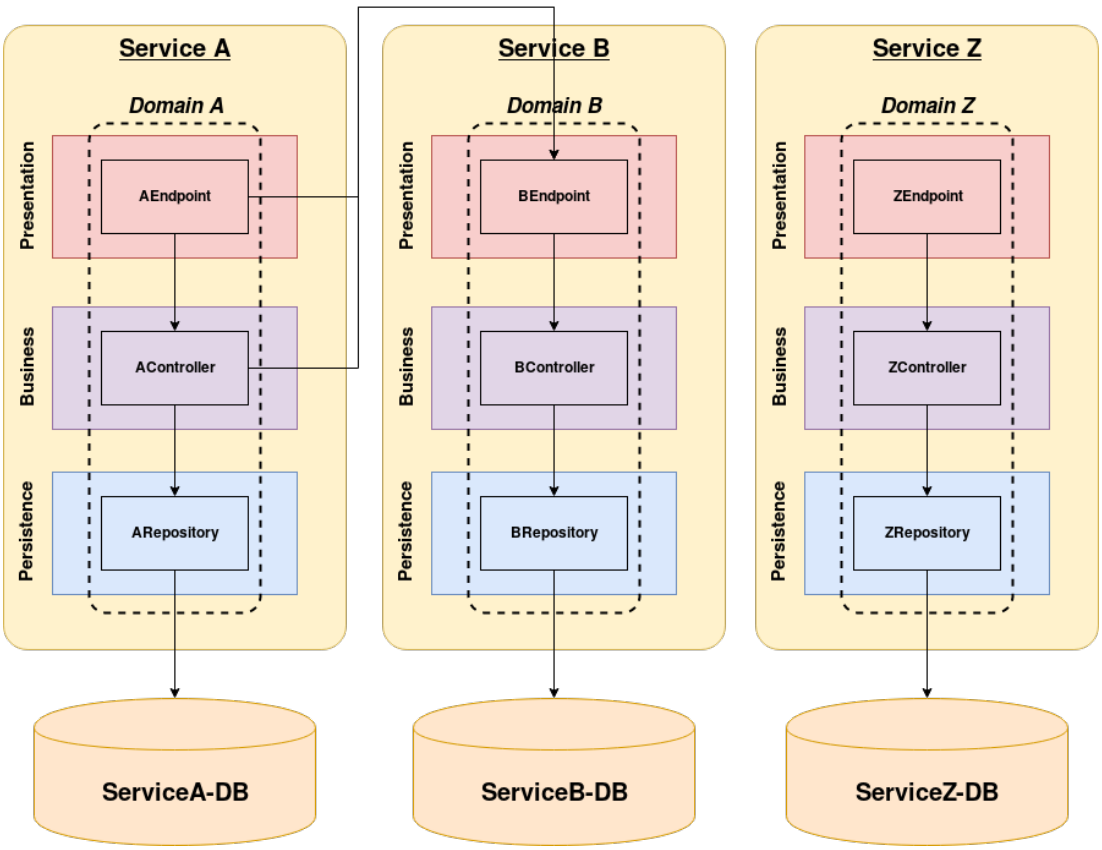


Figure 2.2: Layers in a microservice-based Architecture according to [Sha17]

These two different ways of structuring a system lead to different ways of scaling it. Therefore, a monolithic application is replicated on multiple servers as figure 2.3 suggests. The reason for scaling an application may be a rise of the load on the system. But is it necessary to scale the whole system? In scenarios where a small bottleneck limits capabilities of the system, scaling everything seems not to be a sophisticated way of solving this problem. The microservice-based style makes scaling in such scenarios very effective because specific parts can be scaled without replicating other unnecessary services. [LF14]

This is only one benefit of microservices. Section 2.5 describes more possible gains of a microservice-based architecture. But as already mentioned, communication

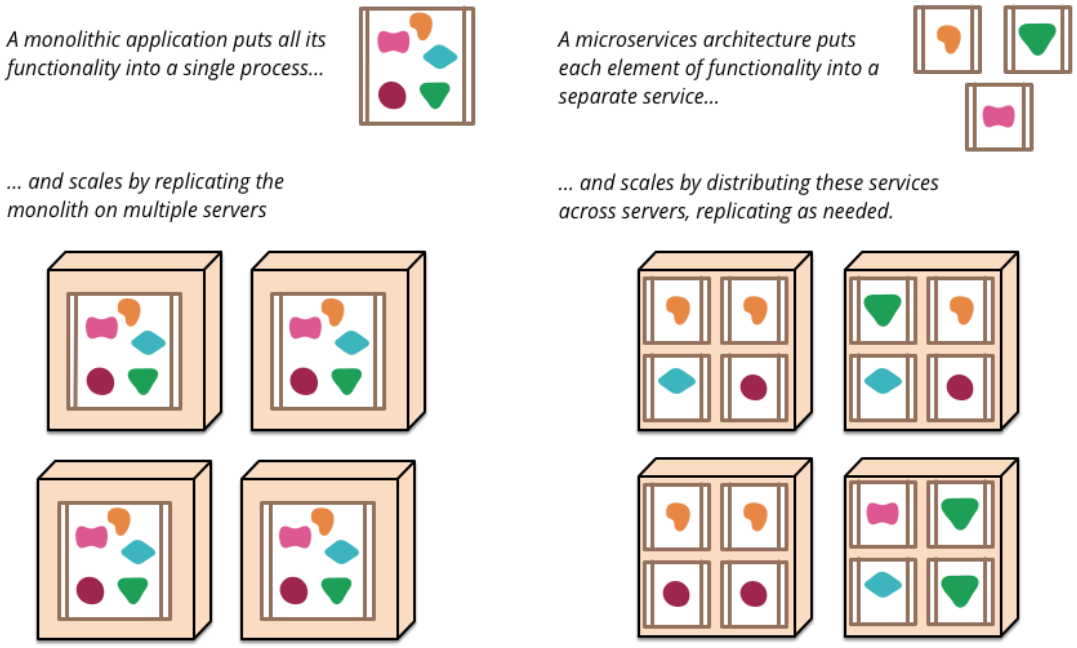


Figure 2.3: Monolith vs. Microservices [LF14]

is not as easy as just invoking local methods. Section 2.6 covers additional challenges introduced by microservices.

2.1.2 Comparison to SOA

There is some discussion on whether microservice-based architecture is not just the same as Service-Oriented Architecture (SOA) - both names already suggest that services are in the focus. Sam Newman defines SOA like this:

> „Service-oriented architecture (SOA) is a design approach where multiple services collaborate to provide some end set of capabilities. A service here typically means a completely separate operating system process. Communication between these services occurs via calls across a network rather than method calls within a process boundary.“ [New15, chap. 1]

This definition sounds familiar regarding the already discussed aspects of microservices. Furthermore, the idea of SOA rose in order to tackle the issues with large monolithic applications, but there was a missing consensus on how to do this in a good way. [New15, chap. 1]

Fowler explains that SOA can mean too many different things, there is no common understanding of this architecture type in detail. [Fow05]

Newman summarizes his comparison by claiming the microservice-based architecture style emerged from „real-world use, taking our better understanding of systems and architecture to do SOA well.“ [New15, chap. 1]

On the contrary, there are opinions that both concepts have major differences. For example, there are differences in the scope. SOA focuses on the entire enterprise IT in terms of the structure while microservices can be applied to only one smaller project. [Wol16, chap. 6.2]

In summary, making a significant comparison between both architecture styles is difficult because neither of them is defined in a standardized way. Many concepts in microservice-based architectures are not novel and can be found in SOA and other architecture styles that existed beforehand.

2.2 Service Boundaries

Designing a microservice-based architecture is not trivial. Especially deciding which parts of the system belong into which service is challenging. Those decisions can have an impact on how the system performs. Especially moving functionality from one microservice to another is expensive due to the strong modularization. [Wol16, chap. 5.2]

Thus, considerations where to put which functionality should be made very carefully. Domain-Driven Design (DDD) can be utilized as a tool to address the mentioned challenges. Furthermore, some data-driven approach is discussed and the connection to Representational State Transfer (REST)ful design is established in the next sections.

2.2.1 Domain-Driven Design

> „Exactly such a model is necessary for the division of a system into microservices. Each microservice is meant to constitute a domain, which is designed in such a way that only one microservice has to be changed in order to implement changes or to introduce new features. Only then is the maximal benefit to be derived from independent development in different teams, as several features can be implemented in parallel without the need for extended coordination.“ [Wol16, chap. 3.3]

The mentioned model is called a *domain model*. It results from the Domain-Driven Design (DDD) that models an „*Ubiquitous Language* in an explicitly *Bounded Context*“. [Ver16, chap. 2]

It embodies a language that is well defined and understandable by the developers and business experts within a context. The domain consists of multiple Bounded Contexts that each have an interface to the outside world. [New15, chap. 3]

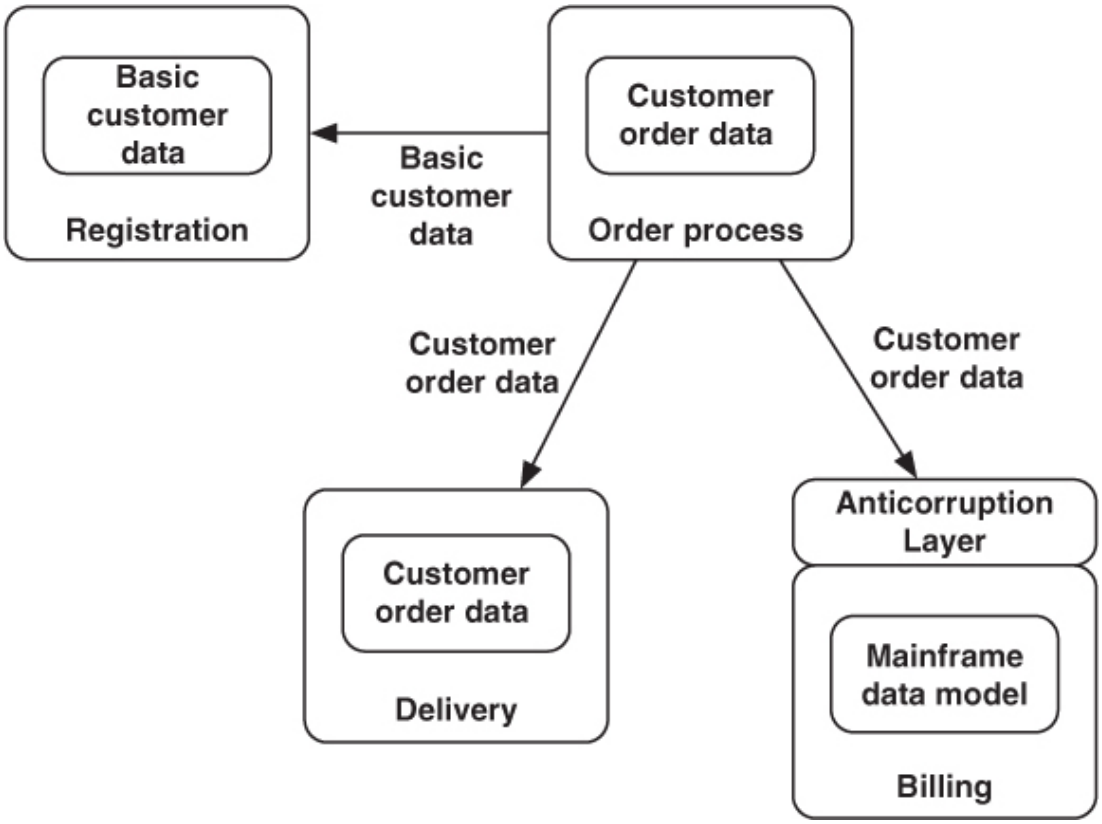


Figure 2.4: Example of a Context Map [Wol16, chap. 7.2]

Figure 2.4 shows an example of connected Bounded Contexts. Each context has its view on the model of a customer. For example, the basic customer data could consist of name and birthday in the registration context. The customer order data extends the customer data with the additional address information for shipping. The billing context uses an anti-corruption layer to decouple the internal legacy model from the interface. More on how to map contexts can be read up in [Ver16].

Like the introducing citation already claims, one microservice is supposed to cover one bounded context. This makes it easier to locate changes for a new feature to exactly one service.

2.2.2 Data-Driven Approach

The JValue Open Data Service (ODS) contains pipelines which process data. Using DDD in this use-case may be challenging if no real domain objects can be identified regarding such a pipeline. But even if this is the case, it may be worth to take a look at this topic from a data-driven point of view.

Data processing pipelines can be split up into steps that are performed sequentially. Depending on the context of the application, each of these steps may be a candidate for a microservice. Such a processing step gets some data as input, does the intended work and provides an output which may serve as the result of the whole pipeline or some kind of interim result that is used as input for the following processing step in the pipeline.

Moving some or all steps into dedicated microservices can provide some advantages in comparison to deploying the whole pipeline as one microservice. If different processing steps benefit from different technologies or programming languages, the microservice-based approach suites perfectly to benefit from that. For example, JSON data can be processed much easier by using JavaScript on a NodsJS server than using a Java library.

Dividing functionalities across multiple microservices as mentioned above comes with its overhead. Network calls can introduce communication overhead and influence the performance in a negative way, especially if huge amounts of data are moved from one microservice to another.

2.2.3 Correlation to RESTful Design

> „One of the first steps in developing a RESTful web service is designing the resource model. The resource model identifies and classifies all the resources the client uses to interact with the server.“ [All10, chap. 2]

A domain analysis based on use-cases can be utilized in order to find those resources from the point of view of a client. There may be considerations on how fine-grained those resources are. For example, the resource of a user could include his addresses, but modeling both concepts as separate resources is also valid. Some design decisions regarding those resources may originate from other considerations. [All10, chap. 2]

If the service boundaries of an existing monolithic system are to be identified, it might be worth having a look at the design of the RESTful Application Programming Interface (API) of the application. Some domain concepts may be directly visible in the resource design which represents each concept with a Uniform

Resource Identifier (URI). Additionally, the dependencies of resources may be visible as well. For example, the URI *users/{userId}/adresses/{id}* shows that addresses depend on users. This information gives a hint that splitting these concepts into different microservices may lead to the necessity to deal with this dependency by API calls or data replication.

A good existing RESTful design seems to be a good starting point to determine the service boundaries. In most cases, it is probably not enough to rely just on that because only the domain concepts visible to the client are included and some resources may exist due to other design choices.

2.2.4 Size of a Microservice

The size of a microservice is affected by the chosen service boundaries. Determining the right size is not trivial because there are many factors that define which size is too big and which is too small. The following list gives an impression about influences on the service size:

- **Team size:** One team should be able to cope with the whole service. This is an upper boundary for the size of a microservice. A team may be responsible for more services but there should never be a shared responsibility regarding one service between teams. [Wol16, chap. 3.1]
- **Modularization:** A developer should be able to understand the whole service and thus be able to develop it further. [Wol16, chap. 3.1]
- **Replaceability:** A service should have a size which makes it not too expensive to replace it. [Wol16, chap. 3.1]
- **Infrastructure:** It should still be possible to provide the infrastructure with an appropriate effort. If this is not the case, it may be a hint that the microservices are sliced into too small components. [Wol16, chap. 3.1]
- **Distributed communication:** There is a notable cost that comes with using distributed communication. This overhead has to be appropriate for the system. Thus, there should be as less interaction over the network between services as possible. [Wol16, chap. 3.1]
- **Consistency:** Strong data consistency can only be guaranteed inside a microservice which influences the minimum size of the individual service. [Wol16, chap. 3.1]

Summarizing, a good size for a microservice leads to the following two characteristics: **loose coupling** and **high cohesion**. Loose coupling means practically that changes in one service should not lead to changes in other services. High co-

hesion groups related behavior together into one service while unrelated behavior is separated by service boundaries. [New15, chap. 3]

2.3 Communication between Microservices

The last section mentioned that communication over the network is expensive and thus should be avoided if possible. But there are scenarios where it is unavoidable that microservices interact with each other. Generally, there are two types of communication:

- **Synchronous Orchestration:** Synchronicity causes the caller to be blocked until the operation is finished and the request is typically answered by a response. The orchestration aspect requires a component that triggers all required actions in the system actively, like calling other microservices. [New15, chap. 4]
- **Asynchronous Choreography:** Asynchronous means the caller does not wait for the request to be completed. This enables an event-based style of communication where subscribers can act on published events. The choreography aspect implies that a part of the system is informed of its job. It reacts by itself with the detailed action. [New15, chap. 4]

Especially when data is required in a microservice that is not within the core responsibility of the service, these ways of communication can be used in order to achieve different results. There are three major ways of dealing with shared data, each one has its strengths and weaknesses. It depends on the specific use case which one suits better.

- **Fetching on demand:** Whenever a microservice requires data from another service, it dispatches the corresponding request and fetches the data. This is a straight forward way of getting the data but slows down the system if multiple services are interacting with each other or a whole chain of calls is triggered.
- **Replication:** Another way is to replicate the required data to your service in the format that is required according to the Bounded Context (see 2.2.1). The replication can happen via an optimized interface or events. [Wol16, chap. 8]

  The advantages and disadvantages strongly depend on the chosen consistency model of the replication. If strong consistency is desired, the data has to be replicated synchronously by using an interface which blocks the system during replication. Asynchronous replication via events or scheduled

interaction with the defined interface leads to eventual consistency but does not raise the latency of the system as much as synchronous approaches.

- **Shared libraries:** Making good use of shared libraries is not trivial, some pitfalls are discussed in section 2.7. The quality of those libraries lies in sharing data that does not change very often which reduces the risk to build a deployment monolith. An example for that would be the names of countries.

2.4 Integration of UI

In most applications, a User Interface (UI) is required in order to make it accessible for users. In the context of a monolithic application, the UI is delivered as a client application that interacts with the server. In a microservice-based system, things might get more complicated. This section specifies three basic ways of integrating the UI into such a system.

API composition

The solution that is closest to the monolithic approach is the API composition shown in figure 2.5. Instead of making requests to the monolith, the UI sends requests to the corresponding services. The existence of multiple microservices may even be concealed by using an edge server described in section 5.3.

This way of integrating the UI is well-known and can be implemented with frontend technologies. However, when changes are made on microservices that break their API, the client has to be adapted as well. This may lead to a deployment monolith like explained in section 2.7. In order to prevent that, API versioning should be introduced. Another disadvantage is that the responsibility for the frontend is not obvious. Is there a frontend team that has to communicate with all the other teams? Or is this responsibility shared, so the team responsible for a certain microservice has to keep the UI up-to-date? These questions have to be answered to prevent inconsistencies between the APIs of the microservices and the use of them in the UI.

Fragment composition

In order to deal with the downsides of the API composition approach, the microservices can serve their own UI components. Figure 2.6 shows that the UI project has only the responsibility to put the fetched components into the right

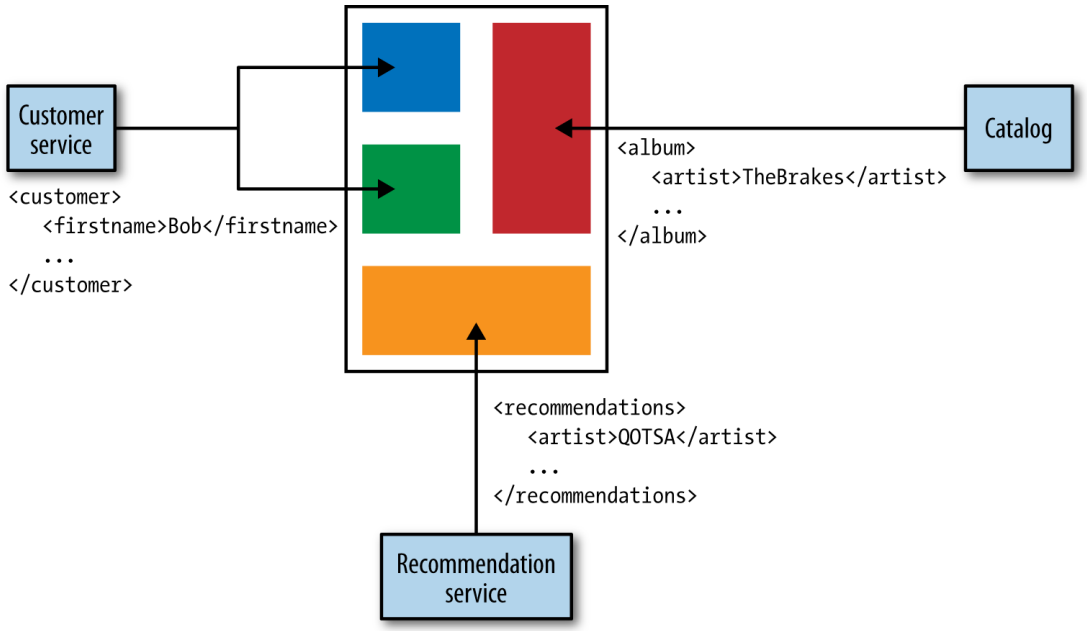


Figure 2.5: UI by API Composition [New15, chap. 4]

place. API changes of the services can include the required changes of the UI components and prevent inconsistencies. The responsibility for the delivered UI components is obviously connected to the team that implements the microservice.

With this approach, new difficulties arise that need to be overcome. How to make the UI look and feel like as if it is from one source even though it is not? What to do if components served by different services have to interact with each other? Which technology does support such scenarios or do we have to write pure self-contained JavaScript components?

Backends for frontends

Finally, the third way of integration is presented in figure 2.7. This solution is orthogonal to the other ones and focuses on how to implement multiple different UIs like mobile apps, web applications and so on. With each UI having its own requirements for the API, either communication overhead would be introduced in order to get the data required, or a coupling would be introduced in order to fit the needs of the clients. The developers may consider implementing multiple backends that provide the APIs that the specific clients require. This approach can be mixed with the ones above.

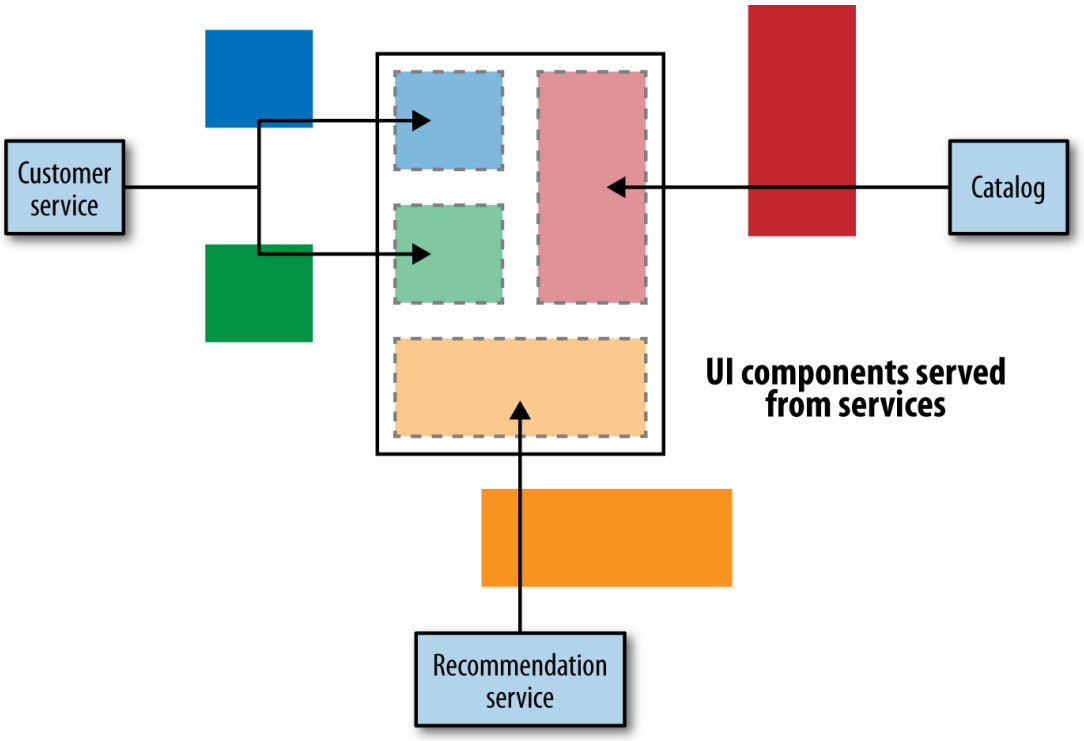


Figure 2.6: UI by Fragment Composition [New15, chap. 4]

2.5 Possible Gains

In order to evaluate if a microservice-based architecture is suitable for a certain project, the possible gains must be clear. It depends on the individual case if these advantages justify the use of a microservice-based architecture. The word „possible“ is here emphasized because those gains are only reachable if the implementation of this architecture style is done right.

- **Scaling:** Microservices can be scaled individually to their needs. This implies there is no unnecessary scaling like in monolithic applications (see section 2.1.1) which in the end may save money. The distribution of the same service to multiple places in the world in the context of a globally distributed system shortens the way between clients and the services and thus may reduce latency. [Wol16, chap. 4.1]
- **Maintainability:** Like in section 2.2.4 summarized, microservices should provide loose coupling and high cohesion. Software design that enables these two characteristics leads to a reliable and more maintainable system. [HM95, p. 1]

  Teams are able to handle the complexity of the overall system better by the

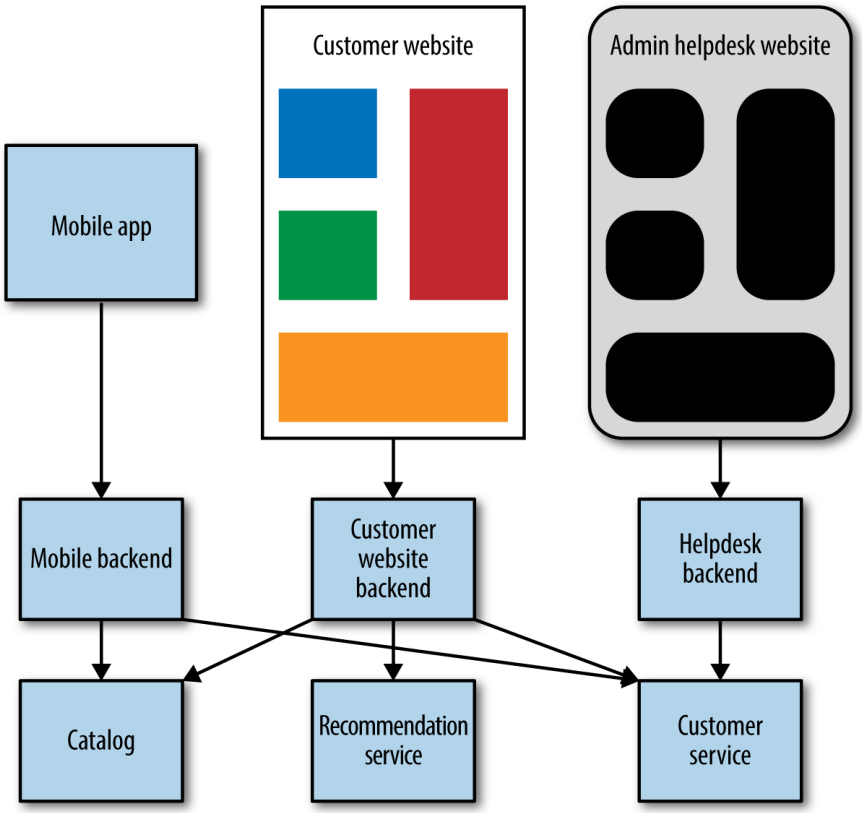


Figure 2.7: Backends for Frontends [New15, chap. 4]

modularization into autonomous microservices. The work on one service does not require to understand the whole system. This implies that the system is also easier to understand. [Wol16, chap. 9.1]

- **Replaceability:** A microservice should have the size so that a complete replacement does not cause too huge effort (see section 2.2.4). Throwing an existing service away and re-implementing it is feasible because, for example, the used technology became outdated.

  Compared to the monolithic approach, it can be prevented to maintain a system that is not reasonable to be maintained. Replacing it instead may save a lot of effort in the long run. [Wol16, cap. 1.2]

- **Robustness:** In a monolithic application programming mistakes like memory leaks can derail the entire application. If this happens to one microservice, only that one is affected and the rest of the system is still operable. This assumes that failures are not propagated and services running on the same hardware are isolated from each other.

  Wolff further claims that design techniques exist so that „microservice-

based systems can tolerate the failure of entire microservices and therefore become more robust than a deployment monolith.“ [Wol16, chap. 4.1]

- **Continuous integration:** Microservices are autonomous and should have individual continuous integration pipelines. This improves the feedback time for developers because only the services are build and tested that were actually changed instead of the whole system. It is recommended to use a separate source code repository for each microservice in order to make responsibilities between multiple teams explicit: a repository together with the build belongs to one team. [New15, chap. 6 and 10]
- **Choice of technology:** Each microservice is generally independent of the others in means of technology choice because the communication is implemented over network protocols. Some technologies have advantages for certain functionalities, so they may be a good choice. Sometimes the developers are used to a programming language or a framework, so it makes sense to go with it in order to be productive. All these choices suddenly become viable when speaking about microservices. [Wol16, chap. 4.1]
- **Organizational structure:** The architecture has an influence on the organization due to Convey's Law: „Any organization that designs a system [...] will inevitably produce a design whose structure is a copy of the organization's communication structure.“ [Con68]

  In other words, teams should form around microservices and take full responsibility. Those teams should be cross-functional because a microservice typically contains the whole stack of an autonomous component. [Fow05]

  The modularization of the whole system into multiple smaller projects with well-defined borders reduces the need for communication between teams that are responsible for different microservices and the risks of the project. [Wol16, chap. 4.2]

2.6 Possible Pains

A microservice-based architecture represents a form of distributed systems. For those kinds of systems the CAP theorem applies which claims that web-services, and thus distributed systems in general, cannot provide the following three guarantees at the same time:

- consistency
- availability
- partition tolerance [GL02]

Distributed systems with no partition tolerance don't exist in practice. So either consistency or availability has to be restricted in any form. [New15, chap. 11]

This brings some possible pains into the project, but also in section 2.5 discussed gains can lead to negative consequences if not implemented in the right way:

- **Consistency:** Usually transactions can be utilized to ensure consistency in a monolithic architecture. Either the whole transaction is performed or everything is rolled back to the prior state. Transactions across multiple microservices are very hard to implement and involve a lot of coordination effort. [Wol16, chap. 3.1]

  Newman suggests designing a system to have weakened consistency guarantees, for example, eventual consistency. This means the data is delivered eventually to all services and until then old versions of the data may be readable. [New15, chap. 11]

- **Availability:** Communication over the network is unreliable. This has to be compensated by the application logic. Calls over the network introduce latency, so synchronous calls between services should be avoided as far as possible. Especially when services are sliced into too small components, you could call them nano-services, latency can increase significantly. [Wol16, chap. 3.1]

- **Service boundaries:** Like section 2.2 discusses, determining the boundaries for microservices is not trivial. It is hard to move functionality from one service to another without breaking the system. So if this step is not done right, future refactorings may come with a lot of costs.

- **Heterogeneity of technology:** The technology choice for a microservice is generally independent of other services like covered in section 2.5. But there still is the tendency towards standardization in a certain way: „sharing useful and, above all, battle-tested code as libraries encourages other developers to solve similar problems in similar ways yet leaves the door open to picking a different approach if required.“ [LF14]

  Summarizing, reinventing the wheel in every microservice is too much effort, so using similar technologies is reasonable in many scenarios.

- **Monitoring:** In monolithic applications logging, debugging and monitoring is straight forward and very common. Introducing a distributed system makes these jobs less trivial. Logs and metrics have to be aggregated in a central place in order to be valuable and therefore should have a common format. Fortunately, there already exist technical solutions for this. Tracing actions in the system becomes harder, because one request to a service may lead to actions on other microservices. This can be approached by

using correlation ids that are passed through the system so everything can be traced in the central logging system. [New15, chap. 8]

- **Versioning:** The independent deployment of microservices is one of the gains that this architectural style can provide. However, sometimes there are breaking changes in the interface of a service that cannot be avoided. In order to not break the consumers, there are techniques like semantic versioning allowing the coexistence of multiple endpoints of different versions within a service or even different deployed service versions. Even though, maintaining this even for a transition phase between versions increases complexity and requires effort. [New15, chap. 4]

Most of the presented pains contain a few points on how to tackle these issues, so they are not unsolvable. Nevertheless, there are some possible pain makers that have to be dealt with when introducing a microservice-based system. The next section discusses a few smells that may increase the pains and reduce the gains of microservice-based architectures.

2.7 The hidden Monolith

During the evolution of a microservice-based system, some pitfalls have to be avoided in order to not build a hidden monolith. This term expresses that independent releases and deployments of microservices are not possible anymore, so you end up with a solution that combines the disadvantages of a monolith and of microservices. There are smells that are commonly found in projects with a microservice-based architecture style. The ones that may lead to deployment monoliths are discussed in this section:

- **Shared persistence:** Sharing a database between multiple microservices is risky because there is no clear data ownership defined. Moreover, even if there would be responsibilities defined, if one service changes the format of the stored data, the other services may crash if they access the changed data. All affected services have to be adapted and deployed together - as a monolithic deployment process. This can be solved by using different databases or at least using tables and schemas that are private to the specific services. [TL18]
- **Shared libraries:** Sharing behavior between services as libraries (or even frameworks) may seem to be an easy approach to prevent code redundancy at first glance, but it couples the microservices tightly together. Adding functionality to those shared resources requires coordination with other teams and if the behavior has to be the same on all services, this leads to a deployment monolith. Keeping the code redundant in each service is

one thinkable solution which leads to unintended maintenance effort if it is carried too far. A shared service that encapsulates this behavior may then be a better solution. Making this decision is always a trade-off and depends on the context of the application. [TL18]

- **API versioning:** Breaking the API in any way leads to failures within dependent services. This can be prevented by introducing versioning of the API, may it be HTTP calls or events or any other interface of the service. The old version has to be maintained until all services moved to the new version and then the old can be removed. [TL18]
- **Processes:** Section 2.5 states that teams should form around microservices and manage their whole lifecycle. If an organization does not adapt to this, the software product may end up as deployment monolith. There is an increased risk of introducing dependencies between microservices if there is, for example, a database team that can modify every service. Those dependencies lead to deployment monoliths. [Wol16, chap. 12.9]

3 Thesis Requirements

3.1 Architecture Concept

The overall goal of this thesis is to evaluate a new architecture approach for the JValue Open-Data Service in order to make the system easier to understand and enable better maintainability. Based on the vision and the requirements of the ODS, some new architecture concept based on the microservice-approach shall be developed.

3.2 Migration Process

A further requirement is to define a migration process that leads the old architecture to the new one. This process should be abstract enough to be also applicable to other contexts. Additional technologies that are required in a microservice-based architecture should be evaluated as well.

3.3 Implementation

The constructed migration process shall be implemented by example. Therefore, a suiting microservice has to be chosen. After the application of the migration, some evaluations should be done regarding the benefits and downsides of a microservice-based architecture. Understandability, maintainability and performance of the system shall be the major considered points.

4 JValue Open Data Service

4.1 Vision

„A world in which consuming open data is easy and safe.“

This is the vision of the JValue Open Data Service (ODS). One challenge towards this goal is that **finding** the right open data for the special use-case is hard. Often **using** and **combining** open data is sophisticated. It produces too much *effort* because there are so many protocols and data formats due to lacking standardization on open data. Additionally, the *quality* of the data is unclear, as well as how *reliable* the data source really is. Orthogonal to these problems there is still a potentially unclear *legal situation*, especially concerning the combination of data from various sources.

Overcoming these problems is surely not easy. Therefore, the ODS has to make **finding** the right data easier by accumulating many data sources, provide important meta-data about them and curating the data if necessary. Crowd-sourcing is the approach of choice for this issue. This implies that adding new data sources has to be easy for users, for example by an easy-to-use user interface. The **usage** of open data is affected by the crowd-sourcing approach as well. It becomes easier because data sources that other users already configured can be reused. Moreover, there must be a standardized way to interact with the system like an API and/or query language for all data sources. Additionally, meta-data about the quality of the data should be aggregated. The lack of reliability of a data source can be compensated by storing the data in a more reliable and available system. The **combination** of open data must be supported by helping the user to assess the risk of potential license violations.

4.2 Current Context

Currently, the JValue ODS covers a subset of the capabilities that are listed in section 4.1. It supports the import of data from some pre-configured data formats like CSV or JSON. Data transformation pipelines can be configured so that they store the transformed data and send notifications in the end. There is a query mechanism regarding the stored data, but it is very restricted. The monolithic architecture makes the application not suited for a huge user base that is required for crowd-sourcing.

There exist two applications that use the ODS in its current state. The first one is **Pegelalarm**, a mobile app for Android phones. Users are notified about the water levels of waters of interest. The second one is a project with industry partners called **Netzdatenstrom**. The ODS serves as an integration platform for different data sources and makes the data accessible via a message broker.

4.3 Current Architecture

The ODS is currently a monolithic application. This means the application is developed, built and deployed as one whole project with one underlying database. Like in section 2.1.1 described, this style of architecture has some downsides. The following architecture descriptions are related to the state of the master branch on the 12.11.2018.

The scope of the following section 4.3.1 does not cover every detail of the project, but the core concepts it is built on. Section 4.3.2 covers the RESTful API and section 4.3.3 outlines the project structure of the current version of the ODS.

4.3.1 Core Concepts

Figure 4.1 shows an abstraction of the current model of the ODS. The goal of the system is to retrieve data from a data source and present it in a processed form to the user.

The typical workflow starts with a *DataSource*. It describes the real-world data source with some additional meta-data. In order to use it, a *DataSourceAdapter* is required. Depending on the type of the data source (e.g. REST interface, FTP server, etc.), the suiting adapter has to be configured by the user. This is usually the first step that is performed by the so-called *ProcessorChain* to import data from a configured outside source into the system.

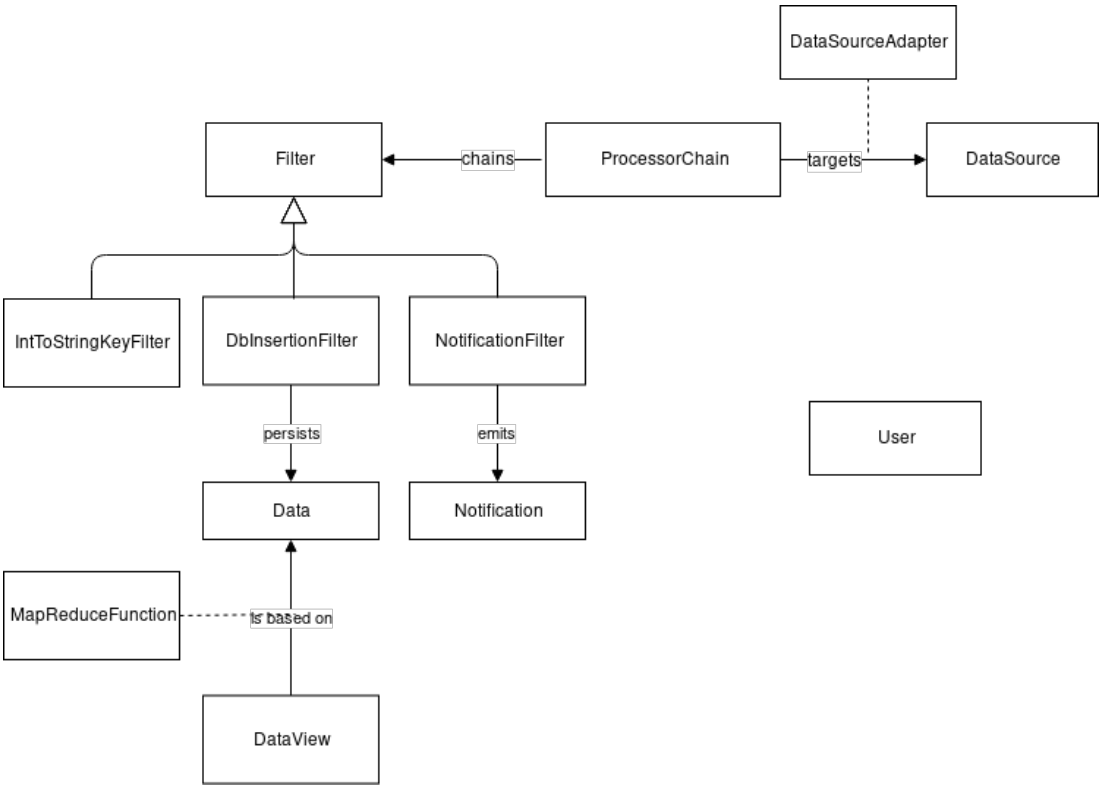


Figure 4.1: Current Model ODS

The orchestrating ProcessorChain can chain multiple *Filters*. They take some data as input, transform or process it in any form and return the result. For example, there is an *AddTimestampFilter* that simply adds the time-stamp to the data. There are two more filters that stand out quite a bit. One is the *NotificationFilter* that does not manipulate data but sends *Notification*s instead. The other one is the *DbInsertionFilter* which inserts the retrieved data as *Data* into the database. Note, that there is currently a filter in development that allows the execution of JavaScript code at run-time. In other words, users will be able to define their custom filters that are executed on the data.

The user can then retrieve this stored data. Additionally, some *DataView* can be set up, a concept that comes with the use of CouchDB as the database layer. Using this database specific feature enables decent performance for combining, sorting and filtering the stored data based on map-reduce-like functions. The DataView can be configured and accessed by the user.

The concept of a *User* exists but is not directly connected to the other concepts in the application. Yet, it is one of the important core concepts because it is required for authentication.

In this section, the important concepts were identified and a typical use-case ODS

was constructed based on them. In the next section, the interface to the outside world is discussed in order to make it clear what users can configure and which functionalities can be added at run-time.

4.3.2 REST Interface

The REST interface represents the way of interacting with the ODS. For the means of understanding the system, an abstract description of the current REST API is presented in this section. A full description can be read up in [Zin]. Especially interesting are the resources (which result in REST-endpoints) that were identified because they correspond to the core concepts that are identified in section 4.3.1. The following resources exist currently:

- /users
- /datasources
- /datasources/{sourceId}/filterChains
- /datasources/{sourceId}/plugins
- /datasources/{sourceId}/notification
- /datasources/{sourceId}/data
- /datasources/{sourceId}/views
- /filterTypes
- /version

Typically, each resource has up to four operations: PUT, DELETE, POST and GET. Those operations may be seen as the implementation of the CRUD-pattern. [Zin, p. 13]

Most of the resources can further be specified by id in order to get detailed information and perform operations on the specific entities.

The *plugins*-endpoint is deprecated and will be removed in the future.

Due to the requirements and goals of the ODS, some API endpoints need special remarks. The first one is the *filterTypes*-endpoint. In order to add filter behavior at runtime, there has to be an endpoint that accepts new filters. This feature is currently in development and will be implemented by providing a generic filter that executes JavaScript code in a sandboxed environment. The second one is the *views*-endpoint which enables creating new views on data via map-reduce-like functions that enables the underlying database layer (CouchDB) to perform queries on the data.

4.3.3 Project Structure

This section discusses the current code-level project structure in order to get an understanding of the structure of the ODS. The following description is on a rather high level in order to get the bigger picture and not to get lost in the details.

```
opendata-service
├── client-retrofit
├── docker
├── gradle
├── models
├── server
│   ├── integrationtests
│   ├── main
│   │   └── java/org/jvalue/ods
│   │       ├── admin
│   │       ├── auth
│   │       ├── data
│   │       ├── db
│   │       ├── main
│   │       ├── notifications
│   │       ├── pegelalarm
│   │       ├── processor
│   │       │   ├── adapter
│   │       │   ├── filter
│   │       │   ├── plugin
│   │       │   ├── reference
│   │       │   └── specification
│   │       ├── pubsub
│   │       ├── rest
│   │       ├── transformation
│   │       └── utils
│   └── test
cep-service
commons
```

Figure 4.2: Current Project Structure ODS

Figure 4.2 shows the source tree of the ODS. There are three GitHub projects that contribute to the ODS in its whole[1]. There are the *opendata-service* project and the *cep-service* project. There is also the *commons* project which provides shared

[1] `https://github.com/jvalue`

libraries between the other two projects. The *cep-service*, short for complex-event-processing-service, is used by the **Pegelalarm** application, which is one of the running systems that use the ODS in production. It sends events to the subscribers depending on the data that the ODS retrieves. These events are dispatched if, for example, the water level rises over a certain threshold. Since this thesis concentrates on the *opendata-service* project, the other two projects won't be discussed in further detail.

Within the ODS project, there is a *models* project. The resulting library contains common model classes for the integration tests and the main project.

In order to deploy the application, Docker is used for containerization. The configurations, including the ones for Docker Compose, can be found in the *docker* directory. The „real“ ODS application is located in the *server* directory, covered with unit tests and some integration tests. The Java classes for the project can be found in the path *opendata-service/server/main/java/org/jvalue/ods*. Most of the identified core classes are also mirrored in the folder structure. But there are also directories that contain code that is shared across these domains, like database access, REST interface, and authentication.

4.4 Possible future Architecture

Before constructing the architecture, the core concepts of the future ODS have to be identified. These concepts implement the vision of section 4.1 and are discussed in section 4.4.1. They lead to the architecture draft that section 4.4.2 presents with all the identified services. Section 4.4.3 covers the design of clients that will be the interaction point for most users of the ODS. Each microservice should implement the requirements that section 4.4.4 suggests. Section 4.4.5 discusses the communication between services. Finally, a workflow of how the services work together to complete a basic use-case is presented in section 4.4.6.

4.4.1 Future Core Concepts

Figure 4.3 shows the core concepts of the future ODS. It is quite similar to the current model described in section 4.3.1 but some generalizations were introduced.

DataSources are now divided into internal and external ones. By supporting the usage of existing Data as an internal source for pipelines, the original concept of DataViews that aggregates data as a view becomes unnecessary. Instead of a view on the data, a new pipeline can be created that takes Data as input and produces Data that fulfills the data aggregation feature as output. This implies that there must be an adapter available that supports access to internal Data.

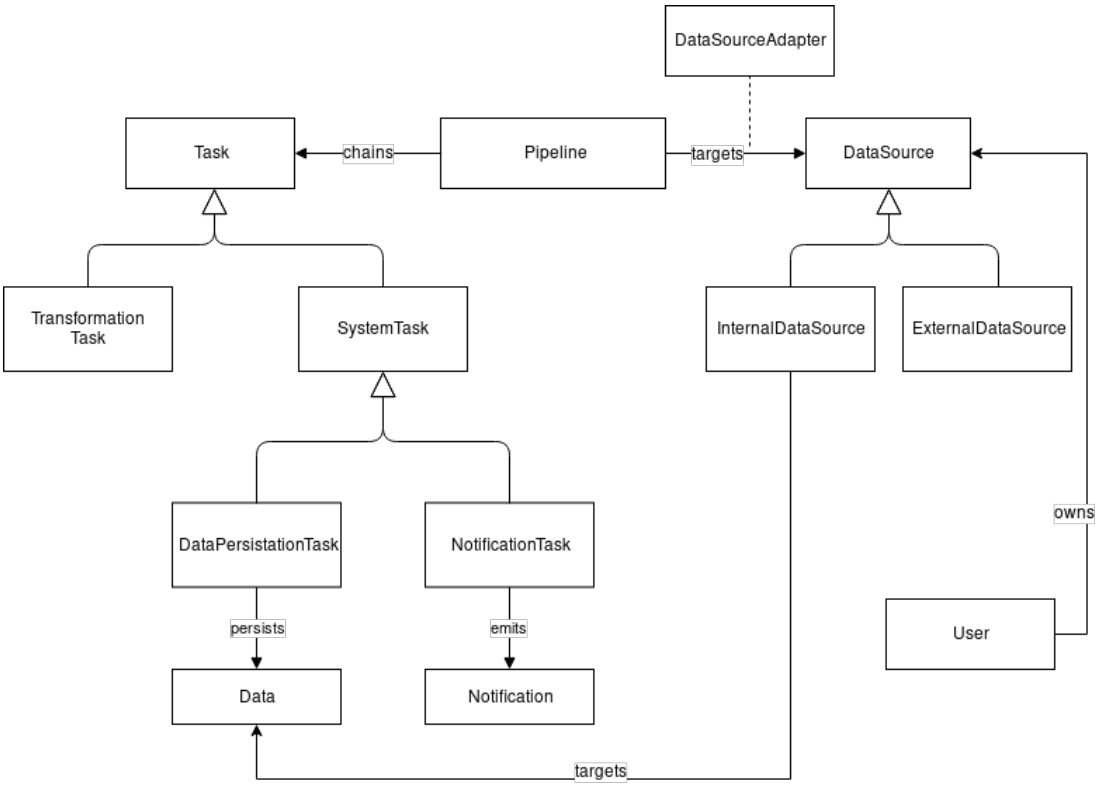


Figure 4.3: Future Model ODS

The concept Filter was renamed to Task because a filter suggests that something is removed or rejected based on some criteria. On the one hand, there are some filters that do a data transformation in the original concepts. But these filters are hard coded and can only be parameterized at runtime. On the other hand, there is the DataTransformationFilter in development which allows the execution of JavaScript code on the data in a sandboxed environment. As an abstraction, all TransformationTasks are defined as JavaScript code snippets in the future ODS and specify the data transformation in a consistent way. Replacing the hardcoded and parameterized transformations by a template system that provides code snippets for common tasks may be an optimization to this approach. It is also thinkable to support additional languages as data transformation languages.

In the original concepts, there are two special filters called DbInsertionFilter and NotificationFilter. These concepts are still present in the future as DataPersistationTask and NotificationTask. They are logically separated from the TransformationTasks as so-called SystemTasks. This conceptual distinction is reasonable because there is no data transformation and they can be executed asynchronously. The further processing of the pipeline does not require a result of these two operations.

4.4.2 Architecture Overview

Resulting from the concept analysis in section 4.4.1, the service boundaries are drawn with the responsibilities. Figure 4.4 shows the identified services.

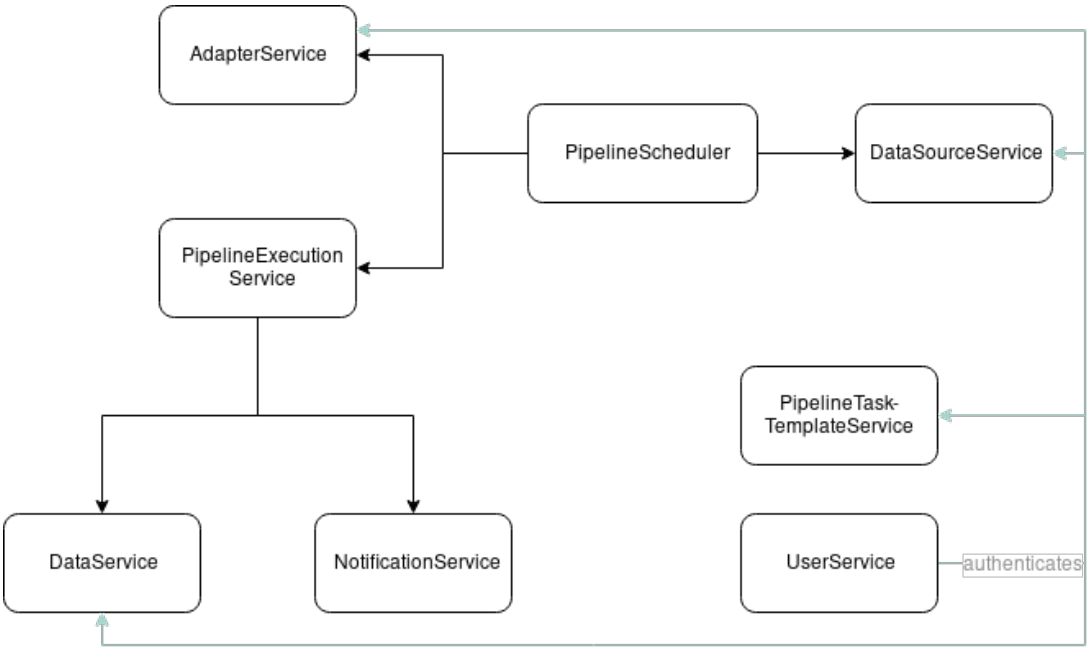


Figure 4.4: Future microservice-based Architecture

- **UserService:** is responsible for managing users and their authentication.
- **DataSourceService:** manages data sources, their configuration, and the pipelines that operate on them.
- **PipelineScheduler:** triggers the execution of pipelines to the defined times.
- **AdapterService:** fetches data according to the adapter configuration of the pipeline. Adapters are implemented at compile-time and are configurable for each pipeline. Creating adapters at runtime is not supported.
- **PipelineExecutionService:** executes tasks according to the configuration of the pipeline.
- **DataService:** responsible for storing the resulting data of pipelines and makes it available to the users.
- **NotificationService:** sends notifications to users in the configured way.
- **PipelineTaskTemplateService:** holds templates for pipeline tasks in JavaScript that can be used as is or modified by the users to define a

pipeline task.

A more detailed description of each service can be found in the documents of Appendix B. Section 4.4.6 shows the typical workflow of all services and their interaction, enabling users to define a pipeline and retrieve data from the ODS.

Figure 4.4 does not show the interaction with clients and how the User Interface (UI) is integrated into the overall architecture. This is a very important topic for its own and is covered in the upcoming section.

4.4.3 Clients Design

The prior section lists all microservices of the future ODS, some more may be added in the progress of the project. This section covers the aspects of the clients that interact with the application. Section 2.4 discusses the basic strategies on how to integrate UI into microservice-based architectures.

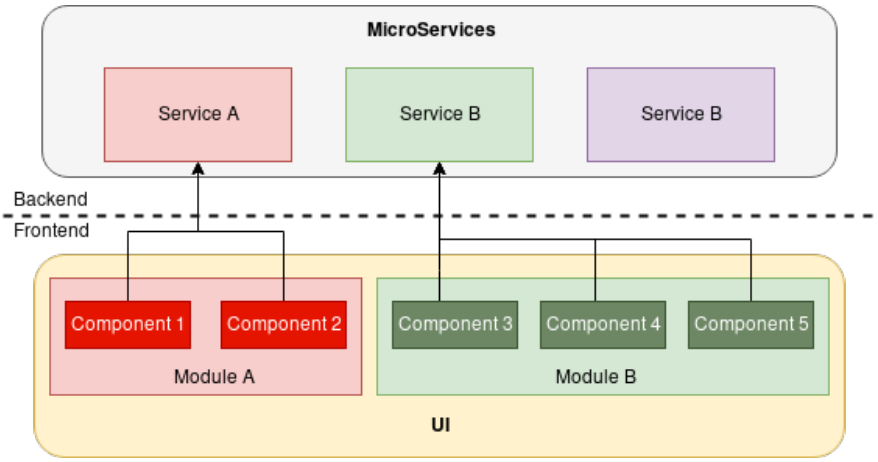


Figure 4.5: UI Integration

Currently, only one UI is planned that serves the users as the interaction point with the ODS. Therefore, the backends for frontends approach is not yet necessary but may be added in the future if needs emerge. The fragment composition technique promises many benefits, but developing a concept benefit from this approach seems inappropriate in these early stages of the project. For the current prototyping-like phase getting results is much more important than introducing a complex, but maybe in the long run better suiting solution that requires a lot of effort to set up. Because of that, implementing the UI according to the API composition is prefered.

However, there are a few recommendations that may help with the issues described in section 2.4. The interaction with services should be encapsulated into

modules that are independent of each other like figure 4.5 suggests. Trying to assign every UI component to exactly one microservice may also lead to a decoupled design and an easy determination of the responsibilities. The communication endpoints of the microservices should be versioned in order to ensure that breaking API changes don't affect the client. The technology used should be as lightweight as possible, support modularization of the UI application and if possible allow the later migration to the fragment composition approach.

4.4.4 Microservice Design

For all the microservices there are a few requirements that have to be fulfilled in order to build a stable system.

- Multiple instances of the services have to be able to run in parallel for scalability of the system. This implies that either the services are stateless, so no state can be lost if the service crashes at any time, or the state can be recovered.
- Consistency across the services is very important. The future architecture targets eventual consistency, so if one service writes data, the other services will eventually know this. Section 4.4.5 states that an event-driven approach is chosen to achieve this.
- The delivery of events follows the at-least-once semantic. This means sent events are delivered eventually for sure, but situations can arise where some events are delivered more often than once. This means that either the services have to filter those duplicates out or always only trigger idempotent actions whenever an event is consumed. Further reading on message delivery semantics can be found in [Pap08, p. 68].
- REST calls to services should not result in multiple synchronous calls to other services in order to achieve low latency. Generally, this means data replication using the events is chosen over fetching data on demand. Of course, this principle is not applied if other approaches are more beneficial in individual cases.
- Semantic versioning of services, their REST APIs and of the sent events should be applied in order to prevent a future running instance of the ODS from crashing due to breaking changes on a single service.
- The deployment of each service must be independent of the other services.
- Accessing data directly from the database, that is under the responsibility of another service, is strictly forbidden. In practice, this means there are no shared database instances between services.

- Each service has to be tested as a black box via integration tests. Especially changes that break the API should be discovered by that in order to prevent the whole ODS to crash.

4.4.5 Communication

Before going into the details of the specific services it is important to know how the communication with the clients or users and between the services will be implemented.

Communication with clients

The communication with the clients is specified as a REST API like in the already implemented version of the ODS. In the future, GraphQL may be considered as an additional interface to the outside world, but this is out of scope of this thesis.

Communication between services

Section 4.4.4 states that the microservices should use asynchronous events for communication between themselves. The idea is to replicate the data a service needs from other services by listening to those published events and constructing the required representation of the data.

There are similarities to the CQRS pattern where multiple read models can be defined that are specialized to the given task in the services for querying. The write-commands are only executed by the service responsible for the specific data. [Fow11]

For example, the PipelineScheduler listens to the events of the DataSourceService and builds up a data structure that fits the purposes of scheduling best. The pipelines could be organized in the service as a priority queue. The next upcoming pipeline that shall be executed is always at the head of the queue. This is much easier to implement than supervising every pipeline on its own. Only the intended execution time of the first element of the queue has to be checked against the current time. If this check is positive, the pipeline can be triggered and the next element of the queue is under observation. The state of the scheduler is represented by the queue, persisting and restoring it is very easy compared to the current approach.

4.4.6 Services Workflow

Section 4.4.2 summarizes the identified microservices of the future ODS. A more detailed description is attached in Appendix B, especially specifications of the REST APIs of the services and which events they produce and consume. The specific data models are left out in order to focus on the understandability of the services and how they interact with each other. This section gives an example of a typical workflow and illustrates the interaction of the microservices.

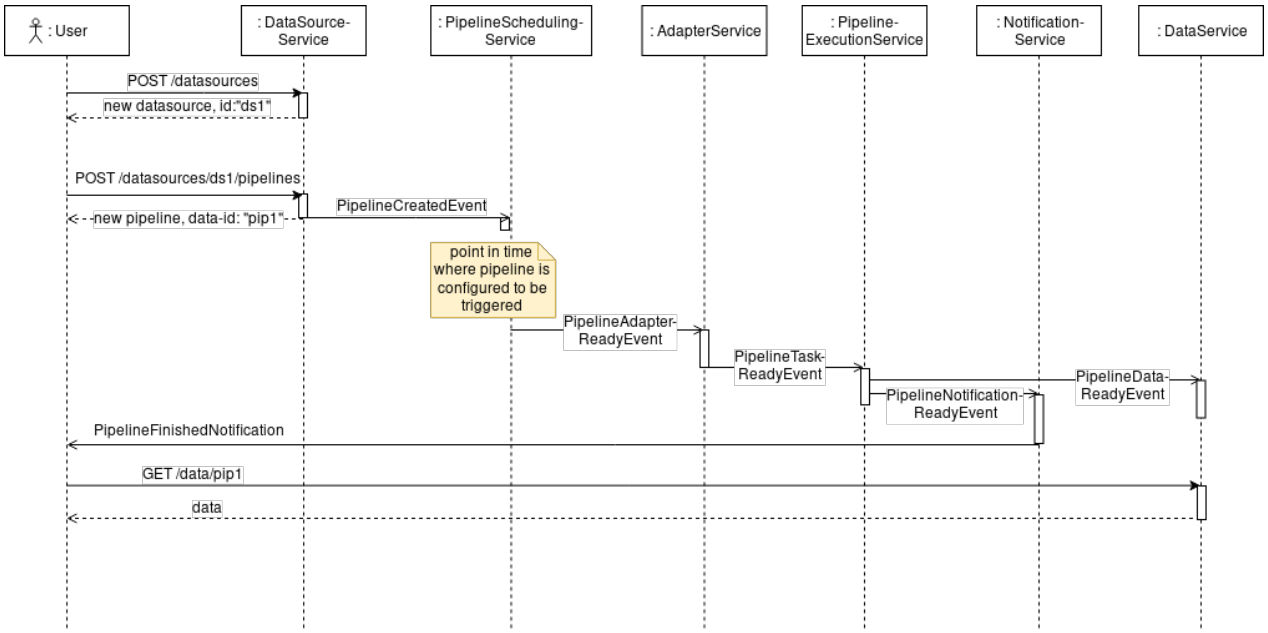


Figure 4.6: Future Workflow

Figure 4.6 shows a typical workflow of a user interaction with the future ODS. First, the user configures the data source and the pipeline via REST calls. This configuration also includes, how and when the pipeline is triggered. The PipelineSchedulingService builds up a data structure that is optimal for determining which pipeline should be worked on next by listening on events regarding the pipelines. Whenever a pipeline shall be executed, it just puts an event on the event distribution mechanism that will deliver it to the right place depending on the pipeline configuration. In this scenario, the pipeline begins with the work of an adapter that fetches data from the defined data source. The resulting data is also published as an event. The following steps are tasks that work on the data. These steps are executed by the PipelineExecutionService. The last part of a pipeline defines the storage of data and a notification to the interested users. The DataService and the NotificationService get the instructions to do their work by events. When the user receives the notification, he can fetch the data via a REST call from the DataService.

5 Migration of the Open Data Service

5.1 Migration Process

Many projects don't have a fixed scope, especially it is common practice in agile projects that requirements change frequently. Thus, architecture evolves over time and has to be adapted to new requirements that come alongside with new features. This means, architecture also changes over time and has to be inspected and rearranged if necessary on a regular basis. For microservices, this includes discussing the service boundaries and their responsibilities repeatedly. In order to do this in a standardized way, there should be a defined process. This process should not be static, but be adapted to the needs of the project and include lessons learned during the process so mistakes don't happen twice. Figure 5.1 shows the process that is used in the scope of this thesis.

First of all, the requirements of a system have to be identified. Particularly, this includes the required consistency of the system. Section 2.6 discusses the trade-off between availability and consistency. This is not a decision that someone should make on the fly, it is actually a decision that has to be made regarding the requirements. This implies that the use-cases of the system should be clear. How many users are supposed to use the system in parallel? Where will the system be hosted and are there any limits for the operational costs?

After these questions are answered, the service boundaries have to be identified. This step is crucial for the success of the project. Rearranging these boundaries in a later stage will be very expensive on the one hand, but on the other hand, it may be unavoidable in a growing project.

Especially at the beginning of the implementation or migration towards a microservice-based system, additional technologies have to be introduced in order to avoid the possible pains described in section 2.5. Additionally, it is very crucial to decide on how the communication between the services will work in the future

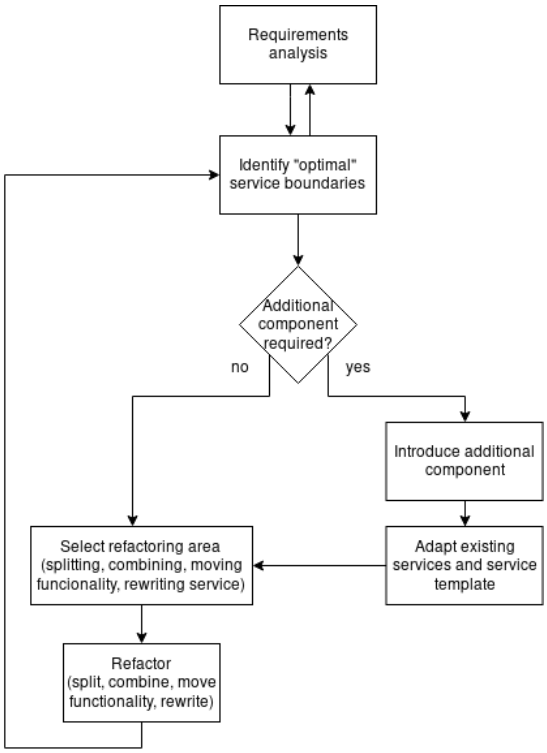


Figure 5.1: Migration Process

based on the analyzed requirements. Additional components and technologies that may be beneficial in such an architecture are covered in section 5.3. If the decision to introduce such a technology is made, it has to be integrated into the existing system. If there already exist multiple independent services, they all may have to be revisited and adapted if required. Another aspect is that existing templates for new services that enable fast productivity have to be adjusted as well.

The next step is to identify which parts of the application have to be refactored in order to get one step closer to the target architecture. Imaginable scenarios are to split one large service into two or multiple smaller ones, merge too small services into one, move functionality from one service to another, and rewrite a whole service. When migrating from a monolith, the splitting process is especially interesting because it defines the step that has to be done naturally most often. Section 5.2 covers instructions on how this can be done. In general, the interfaces should be defined first. Then the refactoring can be performed considering the defined interfaces. The process starts over again after this step.

5.2 Splitting the Monolith

Splitting one large service into two or more smaller services is one of the refactorings that are naturally done very often when migrating from a monolithic to a microservice-based architecture. This slicing should be recorded into a process that is adapted to the project context. The process should be adapted over time by the experiences made during the application of it. In the context of the ODS migration, the process in figure 5.2 is used.

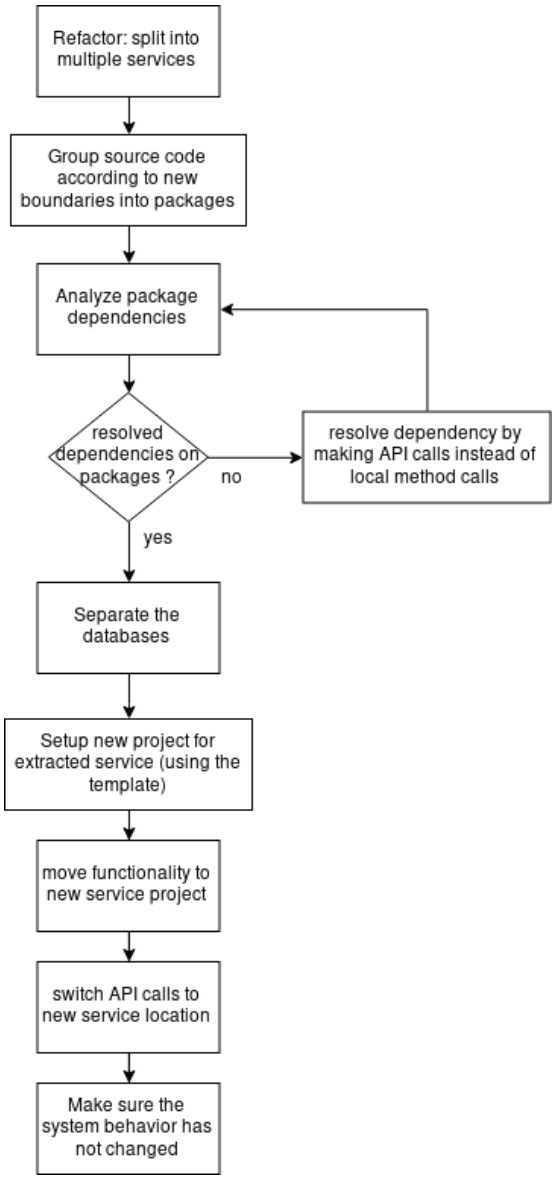


Figure 5.2: Splitting Process inspired by [New15, chap. 5]

The goal is to move a part of the behavior from one service that is to be split into another new service. This does not necessarily mean copy and pasting code from one project into another. Also throwing the old code away and rewriting it, for example in a better suiting technology, is a valid approach.

First of all, the original service has to be analyzed in its implementation structure. Most programming languages offer a logical separation for modularization, so does Java with packages (or Java modules in newer versions). All the code for behavior that is moved into the new service should be grouped in such a module. Most modern IDEs support moving source code between modules and make this refactoring very convenient.

The next steps focus on resolving the dependencies of this module. The other packages have to remove all direct method invocations that target code of the module, except the initialization. Instead, the interfaces to the outside world are utilized to trigger the intended behavior. This may be a RESTful API, some message distribution mechanism or something else. At this point in time, the new interface structure has to be defined and it should match the intended interfaces of the new service as good as possible. In this step, relations between affected tables in the database have to be cut. Some techniques how to compensate that and how to deal with shared data can be read up in section 2.3. Tools which analyze the module dependencies and visualize them can support this process towards low coupling.

Once these dependencies are resolved, the database can be split into two instances. The added database is used to store the data that was decoupled in the former steps. If managing multiple databases in the same project can only be implemented with major additional effort, the separation can be delayed until the new service is created.

Next, the new project can be set up. It is recommended to use a template for this in order to prevent too many different technologies in the scope of the whole projects like section 2.6 warns of. Then the behavior can be extracted from the original service to the new one. If the prior steps were done correctly, the only thing left to do in the original service is to direct the interface calls to the newly extracted service instead of to itself.

As for the last step, the developers have to gain trust in the work they did by showing that the system behavior did not change. Therefore, it is recommended to write integration tests beforehand, so they just have to be run with the new configuration.

This simple sounding process has many pitfalls that should be avoided while migrating to a distributed system. These can be read up in sections 2.6 and 2.7.

5.3 Additional Technologies

There are different migration patterns towards microservice-based architectures. Some of them identify additional technologies that are often used in microservice-based projects and can be found in various publications. They address typical problems in distributed systems and may be considered to be introduced in certain phases of the migration towards a microservice-based architecture.

- **Continuous Integration / Delivery / Deployment:** Automating things is a big part of enabling the potential of microservices. Whenever a developer makes a change in the code, the Continuous Integration mechanism executes defined tasks like building artifacts, running tests etc. in order to provide feedback regarding the changes. Taking steps further, the deployment to a production-like or into the real production environment is desirable if the built artifact turns out to be stable enough. [Bal+18, p. 4-14]

  This automation of infrastructure is one of the key characteristics of microservices. [LF14]

- **Service Registry:** One of the most harmful smells in microservice-based architectures are hard-coded endpoints that are used for service communication. If those endpoints change, the system breaks. [TL18]

  Thus, introducing a component, that makes the endpoint discovery between services dynamic, is recommended. Services register themselves on their start-up at the service registry and can be discovered by other services. [Bal+18, p. 8]

- **Load Balancer:** Service registries already enable the discovery of multiple instances of one service. Balancing the load of the system across these instances in a transparent way for clients is valuable because of a variety of reasons. It „gives us an increased ability to handle load, and also reduce the impact of a single host failing.“ [New15, chap. 11]

- **Circuit Breaker:** If the system fails, the failure should appear very fast in order to avoid unnecessary waiting. Timeouts are one well-known approach that assumes failure after waiting for a certain time. Taking steps further, a circuit breaker enables failing fast without waiting for the timeout in scenarios where multiple failures happened in the past and an upcoming failing operation is very likely. [New15, chap. 11]

- **Edge Server:** It should be unnecessary for clients to know how the internals of the system is built and changes in the internal structure should not affect them. Therefore, an additional front-door layer can be introduced

as the interface for the clients that hides the complexity of the system to them. This component goes well with load balancers and service discovery in order to be highly efficient. [Bal+18, p. 11-12]

- **Containerization:** Making the test and the production environment as similar as possible is desirable for the Continuous Integration, so failures can be found beforehand or if not, reproduced in the testing environment. One lightweight technology that helps towards this goal is containerization. The microservices should be put into containers, which represent the entities that need to be deployed. [Bal+18, p. 12-13]
- **Container Orchestration:** If all services are deployed within containers in a standardized way, some container orchestration tool can handle the details of the deployment. This includes the number of instances per services, some automatic scaling metrics that can start and stop instances on demand, the handling of container failures like restarts etc. This technology can enable scalability and high availability of the system. [Bal+18, p. 13]
- **Logging and Monitoring:** Monitoring the system and its behavior is important. The health of services may lead to actions in the container orchestration, but there also may be other metrics worth monitoring which are able to help developers for future design decisions. Logging is not as easy as in a monolithic application as well. For both aspects, it is desirable to have all the information accessible in one place instead of looking at each microservice individually. Introducing tools that aggregate these pieces of information and process them into a usable format is a common technique. [Wol16, chap. 11.2 - 11.3]

6 Implementation

Migrating the current version of the ODS described in section 4.3 to the future one is an incremental process. Therefore, the migration techniques described in section 5 are utilized. This chapter covers the application of the process in an exemplary manner.

As figure 5.1 shows, the first step of the migration process is the requirements analysis. The results can be found in section 4.4.4. The „optimal“ service boundaries that represent the deliverables of the second step are shown in section 4.4.2 in conjunction with the core concepts in section 4.4.1.

Section 5.3 lists additional technologies. Continuous Integration with tests and Containerization is already present in the ODS project. The Docker containers are managed by Docker Compose, which serves as simple service registry via DNS. For the time being, there are no additional components that have to be introduced. Later on, it turned out that an edge server is required in order to make sure that the behavior of the system did not change. The details are described in section 6.1.4.

Following the defined migration process, a suiting microservice candidate has to be identified for the refactoring. There are various candidates that fit as the first example for this process. The chosen one should be as independent of other functionality as possible in order to guarantee an easy extraction. For example, extracting a part in the middle of the system that has many internal dependencies towards other functionality is not the best option. Those parts are better targeted later on when some dependencies will already be removed due to the extraction of other parts of the system.

The UserService seems to be a suiting choice in this scenario. On the one hand, there are only a few direct calls to this functionality, on the other hand, it will affect all future services due to authentication and authorization. If any other service was extracted beforehand it would have to deal with the distributed authentication anyway that would stick to the monolithic part, so doing this especially deliberately makes sense during this extraction in order to prevent future restructuring of the authentication.

The description of the performed implementation steps in this chapter is written without going into deep technical implementation details in order to keep the focus on decisions regarding architecture and processes. Section 6.1 covers the final step of the migration process, the refactoring.

6.1 Extracting the UserService

Currently, the ODS is a monolithic application. Thus, the splitting process described in section 5.2 is applied. Summarized, this means resolving the dependencies between packages that are targeted to be extracted by refactoring and making API calls instead of local method invocations. Then the databases are separated and the functionality is shifted into the new project for the extracted service. Next, the destinations of API calls are changed to the new service location. In the end, it is important to make sure the behavior of the system has not changed or has only changed in the intended ways. The implementation of these steps is described in the following sections.

6.1.1 Dealing with the Commons Project

Before the actual migration can start, some thoughts must be made about the Commons project of the ODS. Within, there are libraries that typically have a common character across projects of the ODS, which are at the moment the ODS itself and the CEP-Service.

For example, the database access abstraction is located in the Commons project, which may be useful also to other services which are using the same technology. The authentication is outsourced to the Commons project, but unfortunately in a way that it is not very suitable for future use. Probably the authentication mechanism will have to be changed due to its distributed form. But changing it in the Commons project also means that the CEP-service is affected. This is somewhat not desirable.

For the extraction of the UserService, this matter is solved by copying the code regarding the authentication into the project and removing the dependency to those parts of the Commons project. After the extraction, it has to be decided which parts of the distributed authentication may be shared between services and if some shared code across these services is desired or not. Section 2.7 already shows that shared libraries can introduce some downsides, so it is at least questionable if such an approach should be taken in the future. Additionally, with the technology heterogeneity that microservices can introduce, the benefits of libraries are reduced because they usually cannot be used by multiple programming

languages.

6.1.2 Resolving Dependencies between Packages

The ODS project is written in Java. With version 9 of Java SE the Jigsaw project was introduced, which enabled a better modularization of Java projects by defining clear boundaries for modules and their exposed interfaces. To the time this thesis was written, the used build tool Gradle[1] did not support this at a sufficient level, so Java packages were the go-to technique of modularization in the application. After the authentication sources of the Commons project were copied to the project like explained in the previous section, all classes that were supposed to be moved to the new extracted UserService were refactored in order to be located in a new package named *userservice* .

Due to the *find-usages* functionality most modern IDEs offer, it was quite easy to see all dependencies that use the classes within the *userservice* package. The usages of model classes were substituted by copying the used classes and using these copies within all the other packages. The direct method invocations of classes of the userservice package were replaced by API calls on the UserApi. This also includes serialization and deserialization of model classes which made the usage of the copied model classes possible.

During this refactoring, the authentication mechanism was divided into two parts. Only the UserApi makes use of the look-up mechanism operating on the database that checks if the pair of username and password is present or not. All other REST endpoints check the existence of the user by calling the User-API.

After these steps, all direct dependencies towards the functionality of the future UserService were removed. This was also verified by utilizing the *find-usages* functionality of the IDE.

6.1.3 Extracting the new Service

The next step was to move the grouped classes and their functionality to a new project. Therefore a new project in the same Git repository was created, with the same technology as the monolith. A Gradle project that builds a Java project with Dropwizard[2] as framework supplying everything to build RESTful web services was set up. All important dependencies were included and an additional database instance was created. The UserService project was configured to use the new CouchDB instance instead of the old one. The grouped code of the monolith

[1] `https://gradle.org/`

[2] `https://www.dropwizard.io/1.3.8/docs/`

was moved to the new UserService project. In order to make it run, some further configurations regarding Dropwizard and authentication had to be applied. The destination of the API calls towards the UserApi in the monolith was changed to a new instance of the UserService. Because the API itself was not modified, no more changes in the monolith had to be done at this point of time.

6.1.4 System-behavior Consistency Check

As the last point of the migration, the behavior of the altered overall system had to be checked by ensuring the integration tests were still passing. Therefore, the UserService was containerized with Docker[3] like the monolithic project. The deployment was realized by Docker Compose[4]. The additional CouchDB instance was added as well as the built Docker container of the UserService.

Due to the fact that the integration tests were testing against one host, those tests were still destined to fail. The new architecture has two different hosts, one for the requests regarding the UserService and one for the rest. In the end, this was solved by adding a proxy server that routes the requests regarding specified rules to the right running web application and hiding the existence of multiple web services to the outside world. As the technology of choice Traefik[5] was selected as a lightweight edge router. This resembles the edge server of the additional technologies specified in section 5.3. In the future, this may be enhanced or replaced regarding load balancing and service discovery.

After fixing a few remaining minor bugs that the integration tests revealed, the state of passing integration tests was achieved. For the last step, the Jenkins[6] Continuous Integration configuration was enhanced to also build, test and containerize the UserService.

6.1.5 Performance Optimization

How communication between services can work is described in section 2.3. For the authentication scenario, some different approaches were evaluated.

- **Direct API calls:** This is the version that was implemented directly after the extraction of the UserService. Whenever a client requests the functionality of a service, the given authentication header is redirected to the API of the UserService checking if such a user exists. This means every API call

[3] `https://www.docker.com/`
[4] `https://docs.docker.com/compose/`
[5] `https://traefik.io/`
[6] `https://jenkins.io/`

has additional latency added by the call over the network to the UserService. This is something undesirable, so choosing another approach should be considered if it does not introduce any other significant downsides.

- **Caching Users retrieved by direct API calls:** This method is just an optimization regarding the prior version. Users are cached within services for a configured amount of time, so repeated calls don't add latency due to additional authentication calls. In relation to caching, the topic invalidation rises up. What if a user gets deleted and is stilled cached on other services as a viable user? There are scenarios where this is acceptable behavior, depending on the amount of time the users are cached. By listening to events that imply that a user was updated or deleted this duration can be reduced by invalidating the corresponding entry in the cache.
- **Replicating the Users:** In an event-driven architecture, it is easy to replicate the user data to other services by listening to the event stream. Then each service can check if the user is registered or not. This has the advantage that no network latency is added on direct API calls. But this means also distributing the credentials across all services. This doesn't seem to be a sophisticated solution due to security issues, which is its very own topic in the world of microservices. Although the password would be hashed, this may cause security issues in the future.
- **Using internal tokens:** Using internally generated tokens for authenticated users that signed-in is another common approach. An example of this is making use of tokens called JSON Web Token (JWT)[7]. A cryptographical mechanism secures information about the user inside a token. A service retrieving such a token from a client can validate it without making another API call to the UserService but by performing a cryptographical operation. The only thing to do is to retrieve the public key once from the UserService and use it for the decryption of the tokens. This is an established approach with many advantages, but it introduces additional complexity in a software system, e.g. implementing a strategy to invalidate tokens.

After evaluating all the up- and downsides of these approaches, the second one was chosen over the others. The users are cached by the microservices after successful authentication is performed once by the UserService. The invalidation is implemented by listening on events published over RabbitMQ[8], a lightweight message broker. Some retry mechanism for the REST calls to the UserService was introduced in case the network connection fails. The reason this approach was chosen instead of the JWT tokens that offer many advantages is to keep the changes as less invasive to the existing system as possible. However, in the future

[7] `https://jwt.io/`
[8] `https://www.rabbitmq.com/`

it might be worth considering using tokens instead of the current approach.

A performance analysis of the first three approaches supports this decision. For the setup Docker Compose started the required components on one host, so there was no real network between the services.

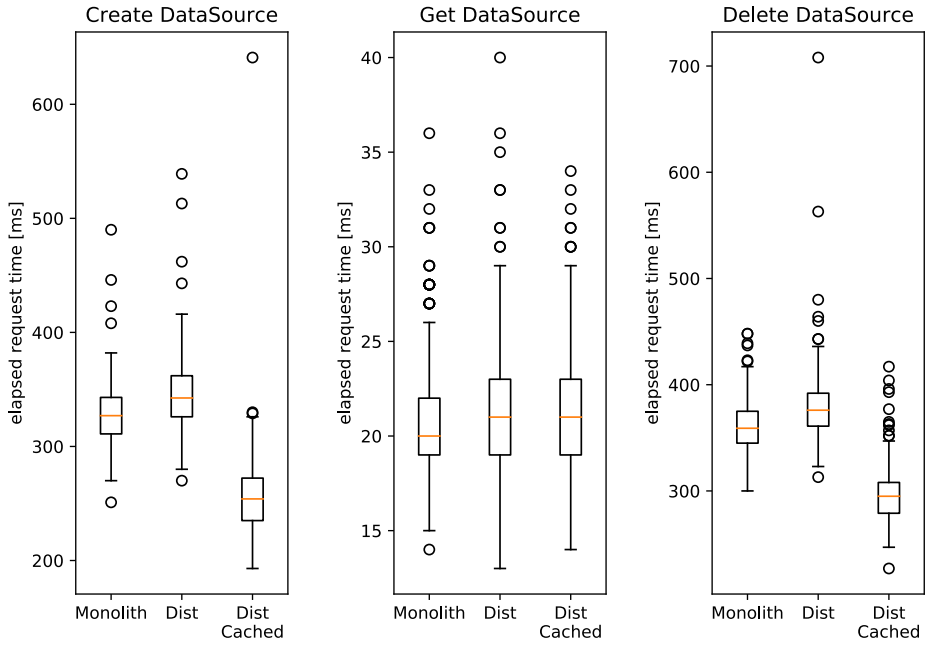


Figure 6.1: Performance of Authentication Strategies

A JMeter[9] script was created that performs the following steps for 1000 times.

- Create a DataSource.
- Fetch the created DataSource.
- Delete the created DataSource.

The elapsed time for the request was measured for three scenarios.

- Monolithic application
- Distributed application with extracted UserService, which authenticates the user on every request
- prior scenario with a caching strategy, so only the first call authenticates the user directly by the UserService

[9]https://jmeter.apache.org/

The results are shown in figure 6.1. The analysis does not include the first half of the measurements to make sure that the influence of JVM warmup and other factors of the starting process of the service(s) are as small as possible.

Create (Monolith)	327.0 ms
Create (Distributed)	342.5 ms
Create (Dist+Cache)	254.0 ms
Get (Monolith)	20.0 ms
Get (Distributed)	21.0 ms
Get (Dist+Cache)	21.0 ms
Delete (Monolith)	359.0 ms
Delete (Distributed)	376.0 ms
Delete (Dist+Cache)	295.0 ms

Table 6.1: Median of Performance Analysis

The results prove the expected behavior. Comparing the creation and deletion on the monolith to the distributed version, it can be observed that the additional network traffic introduces a communication overhead with a latency of approximately 16 ms. This resembles the delay by the network which is rather low because all services run on the same machine in this test. The caching implementation provides a lower elapsed request time because after the first request the system does not require additional network interaction. It is even faster than the monolithic approach. This emerges from the fact that the monolith performs a lookup operation fetching the user from the database for every request. In comparison, the distributed caching version provides the results faster because the data already resides in the cache. For fetching the data sources, no major differences between the implementations can be observed because there is no authentication required for this endpoint.

The analysis shows that the chosen approach with an extracted UserService and caching by the other services is even faster than the monolithic implementation without additional caching. This validates the chosen solution.

6.2 Rewriting the UserService

The defined migration process in chapter 5 shows that discarding existing services and rewriting them from scratch is a valid way of refactoring. Rewriting usually means reevaluating the programming language and framework to make the right choice which one suits best for the corresponding service. But a crucial aspect

is to keep the interfaces to the outside world as they are. In the simplest case, this exclusively concerns the REST interface. The whole rewriting process is an additional effort that brings no business value at first glance but potentially in the future. Features may be implemented faster in the future by making use of special language features or libraries, also the non-functional requirements like performance may be improved by such a choice. The reasons in this context for rewriting the UserService are discussed in the next section.

6.2.1 Chosen Technology

The solution using Java and Dropwizard as the framework was unsatisfying because many common problems had to be solved by implementing it by ourselves. This includes, for example, the access to the database which was abstracted by hand in order to make databases interchangeable. Most of the authentication was implemented by hand. Those mechanisms were located in the Commons project in order to share it across multiple projects and also across microservices in the future. There was much to configure manually, which can be an advantage or a disadvantage. The Dropwizard framework opened many possibilities for custom implementation, but this also requires time and effort. There are frameworks that provide such solutions out of the box and also support the accessibility of additional technologies as they are listed in section 5.3.

The chosen technology for the UserService that should be rewritten was still Java as the programming language. It proved to be reliable in the past and was already well-known by the working developers. Instead of Dropwizard, the framework Spring Boot[10] was chosen due to a variety of reasons. It was even chosen as the most popular application framework in 2018. [Sch]

One major point was that Spring has a huge community and there exist many tutorials and documentation on the web. Due to its popularity there are many extensions that provide common functionality out of the box. Especially interfacing with some implementations of the already mentioned additional technologies of section 5.3 is supported by plugins. Spring Boot provides auto-configuration to the Spring framework which enables a very fast start into implementation and reduces the complexity of configuration. But there is a downside that has to be accepted when choosing this framework. Very specific configurations that are far away from the default way of doing it are hard to apply. Microservices try to reduce the complexity of the given services because they are only responsible for a limited set of functionality. So taking this risk seemed to be acceptable, because the default pre-configured way should work out in most cases.

This technology choice is supposed to form a template for future microservices

[10] `https://spring.io/projects/spring-boot`

in order to prevent a too high variety of technology in the system. If in a specific use-case the explained downside should become too painful, another technology may be chosen for that service implementation.

6.2.2 Technology in Practice

In the context of the UserService, the framework SpringBoot turned out to be a good choice. All dependencies to the commons project could be removed because the required mechanisms are provided by the framework. While programming with the framework it felt more productive due to the auto-configuration. Using environment variables in the configuration files that SpringBoot provide out of the box, together with using different configurations by profiles for deployment and local uses, the whole containerization process became a lot cleaner.

As part of the rewriting, the underlying database of the UserService was exchanged. CouchDB was replaced by MongoDB which was already planned for a while for a few parts of the ODS. This is well supported by SpringBoot, it only took a few lines of code to implement the intended functionality.

But also the negative aspects that come with the auto-configuration were experienced. For authentication, cookies were enabled by default which led to some undesired behavior during testing. Finding the cause for those unexpected interactions was time-consuming.

A performance analysis unveiled that the implementation with SpringBoot and MongoDB performs even worse than the implementation with DropWizard and CouchDB. To make a comparison of both applications possible, the mechanism for sending events and communication endpoints had to be adapted. The following steps were executed repeatedly for 1000 times, the first half of the iterations were not taken into account to reduce the implications of start-up artifacts as much as possible.

- Create user.
- Get the created user.
- Delete the created user.

Figure 6.2 shows that in all three scenarios the SpringBoot implementation is significantly slower. Even though, the development team decided in favor of SpringBoot because of the improved development experience and the enhanced development speed.

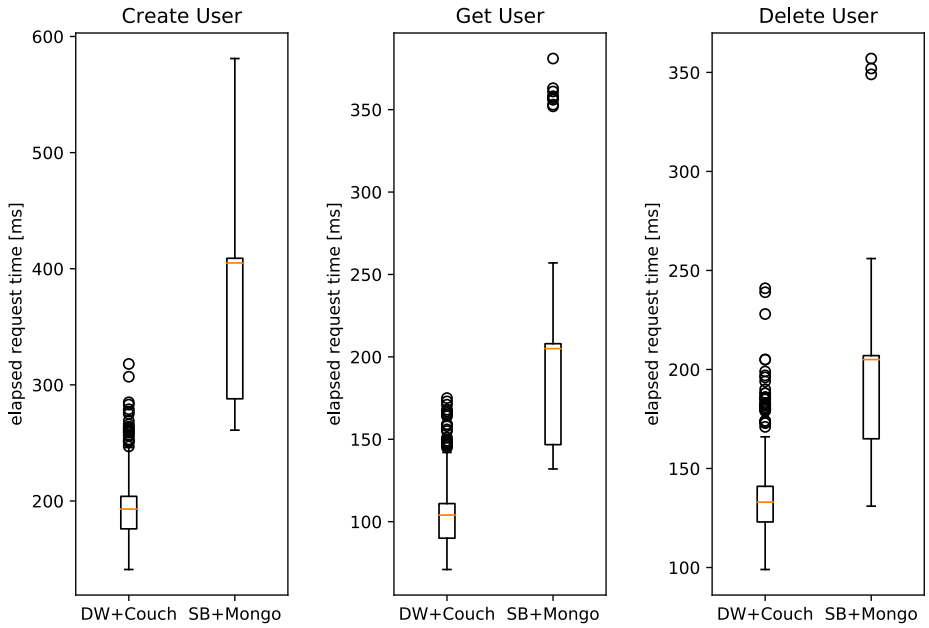


Figure 6.2: Performance of Frameworks

6.3 Dealing with Consistency

Achieving the desired consistency level in an application is far from trivial. Consistency has to be defined between services, between the instances of the same service type and also between clients and the whole system. During the implementation phase, a scenario was discovered where inconsistencies between services and thus, between clients and the whole system can appear.

6.3.1 The problematic Scenario

Section 4.4.4 states that eventual consistency should be achieved in the system. In the future architecture of the ODS, this means that all writing actions on services must evoke a resulting event that is eventually delivered by a message broker. For the implementation, this means conducted writes on the database have to lead to an event that reaches the message broker. The configuration of the message broker ensures the eventual delivery of the event. Rephrasing this, writing on the database and sending events have to be an atomic operation. Scenarios, where only one of both actions is executed, have to be prevented in order to keep the system consistent. The following implementation does not yet

provide solutions to the problem but shows how easy it is to run into this pitfall.

Listing 6.1: Sequential commands

```
...
db.write(user);
broker.publish(new Event(user));
...
```

The problematic scenario causing inconsistencies can arise if the connection to the message broker cannot be established. In distributed systems, the network cannot be assumed to be reliable, so this is a realistic scenario. It would lead to a database write, but the event gets never published.

Trying it the other way around by sending the event first and writing to the database afterwards leads another undesired inconsistency in certain cases. The event gets published, but the write to the database is not successful, for example also due to network issues. The sent event cannot be revoked any more and leads to an inconsistency in other services.

Listing 6.2: Database transactions

```
...
Transaction transaction = db.getTransaction();
transaction.start();
db.write(user);
boolean publishSuccess = broker.publish(new Event(user));
if(publishSuccess) {
    transaction.finish();
} else {
    transaction.rollback();
}
...
```

In this scenario, the database writes get reversed if the publishing to the message broker fails. The inconsistency described above cannot happen in this way. But there is another problematic scenario. This time, the message broker is reached by the publish request, publishes the message, but the acknowledging response gets lost in the network. This means the event got published, but due to the lacking response from the message broker, the database transaction is rolled back. Again an inconsistency found a way into the system. Sending the event first leads to an equal effect as using sequential commands.

Two-phase commits for publishing the event only shifts the problem to the next layer. The atomic way of combining these actions seems very hard to achieve or even not possible. Section 4.4.4 emphasizes that the delivery of events has

to happen according to the at-least-once semantic. This means also publishing events more often than once is basically viable. Also not trying to force those two actions into an atomic command is viable, as long as eventually both are executed. By looking at the consistency requirements from this point of view, new possible solutions arise. In the next sections, two patterns are introduced that provide a solution to the problematic scenario by making use of that strategy. But those solutions come with a certain cost: complexity!

6.3.2 Listen to Yourself Pattern

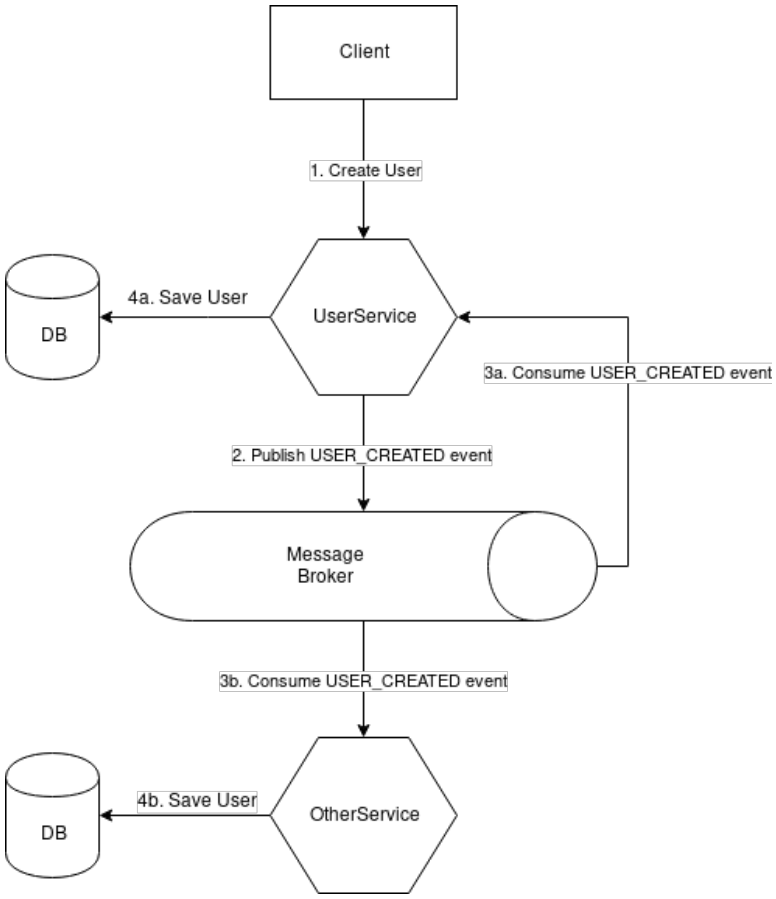


Figure 6.3: Listen to Yourself Pattern based on [Sho17]

The Listen to Yourself Pattern ensures that publishing an event and the corresponding database write are both performed or none of them. Figure 6.3 gives an overview of the pattern. [Sho17]

The service does publish a *WRITTEN-event*, in this context the UserCreatedEvent, on client write-requests but unlike the expectation, writing into the

database does not happen right away. Instead, the service subscribes to its own published events and conducts the write as soon as the event is received again. In parallel, other services can subscribe to those events and react accordingly as if the operation already happened successfully.

Two things can happen in this scenario. Either the event reaches the message broker, then it is guaranteed to be delivered eventually to all subscribers, or it is not published, so the database will never execute the write. Even if the service crashes after the publishing, it will receive the event as soon as it is available again or another instance of the service can take over. The message broker is configured to only remove events from its queue if the processing of it is acknowledged. In a scenario, where the database write is executed but the acknowledgment of execution does not reach the message broker, it will be delivered again. This behavior is satisfying the at-least-once delivery semantic that is aimed to be implemented by the ODS. This means the services have to deal with events that are delivered more than once. Idempotent actions that react to events are a way to ensure this like described in section 4.4.4.

In cases where the write to the database fails, a so-called compensation event may be published which all interested services react to. The design of compensation events is also known under the Saga pattern. Especially in cases, where multiple services react to the creation of an entity that can fail due to conflicts, this pattern brings benefits. Whole event chains can arise, triggered by one original event from user interaction. If one of those fails, the whole action has to be canceled, but some writing actions already happened from events prior in the chain. These services can undo their actions by reacting to such a compensation event. [Sho17]

In figure 6.3, it is visible that the client triggers the action and does not get any response with information on whether the write was successful or not. In the context of the ODS, this is rather an undesired behavior. A client should know if requests succeed or fail. Additionally, clients may not be able to read their writes immediately, which is also a behavior that may be avoided if possible within the ODS.

6.3.3 Application Publishes Events Pattern

The Listen to Yourself Pattern solves the consistency issue by publishing the event first and afterward executing the write action on the database. There also exist approaches that work the other way around, such as the Application Publishes Events Pattern. [Ric]

Each service manages an additional table called Outbox which contains all events that the service produced. Unlike the other approach, the events are not pub-

lished by the service itself. The responsibility lies within an additional application that crawls this additional table and publishes the event after they were inserted. Figure 6.4 shows that this works on the database level. Approaches that enable the fetching of events by a REST API are imaginable as well.

[Ric]

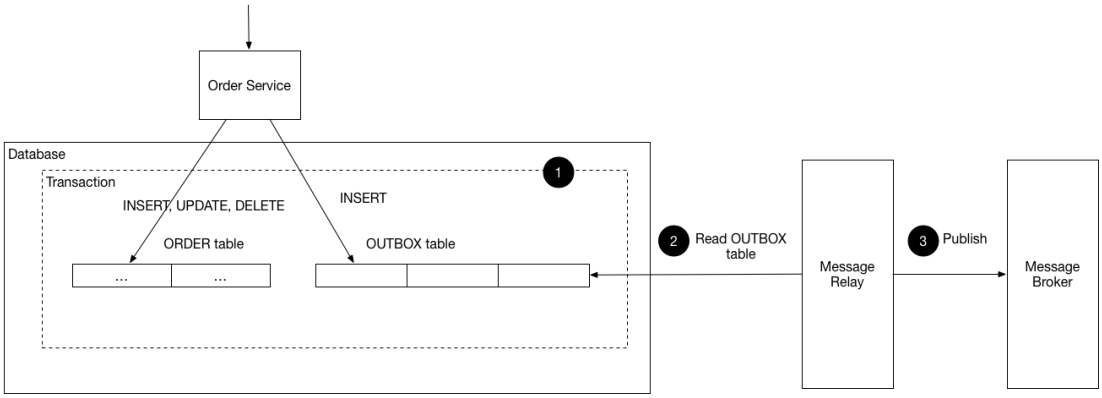


Figure 6.4: Application Publishes Events Pattern [Ric]

The pattern writes the domain object and the event into the database by using transactions. This ensures that both database operations happen in an atomic manner, always both or none are executed. At this point, the client could receive feedback if the action was successful. Due to the outsourcing of the publishing mechanism into another application, the events are published eventually. Even if the service or the publishing application crash, their state is always persisted and can be recovered. Scenarios, where events are published more than once, can arise. For example, if the publishing application publishes the event but does not get a response from the message broker or crashes before the index marking the last published event can be persisted. After a restart, the same event may be published a second time, which is as already mentioned acceptable.

This pattern can be condensed into one application. The separation between the service that writes to the database and the application publishing the events is a logical separation.

There exists a related pattern which operates not on an additional table in the database but on the transaction logs produced by the database. The disadvantage of this approach that reading the events from the log is database specific. [Sho17]

6.3.4 Pattern in Practice

In order to solve the general consistency problem described in section 6.3.1 the two patterns explained in sections 6.3.2 and 6.3.3 are qualified. They were both implemented in order to evaluate which one fits better into the context of the ODS.

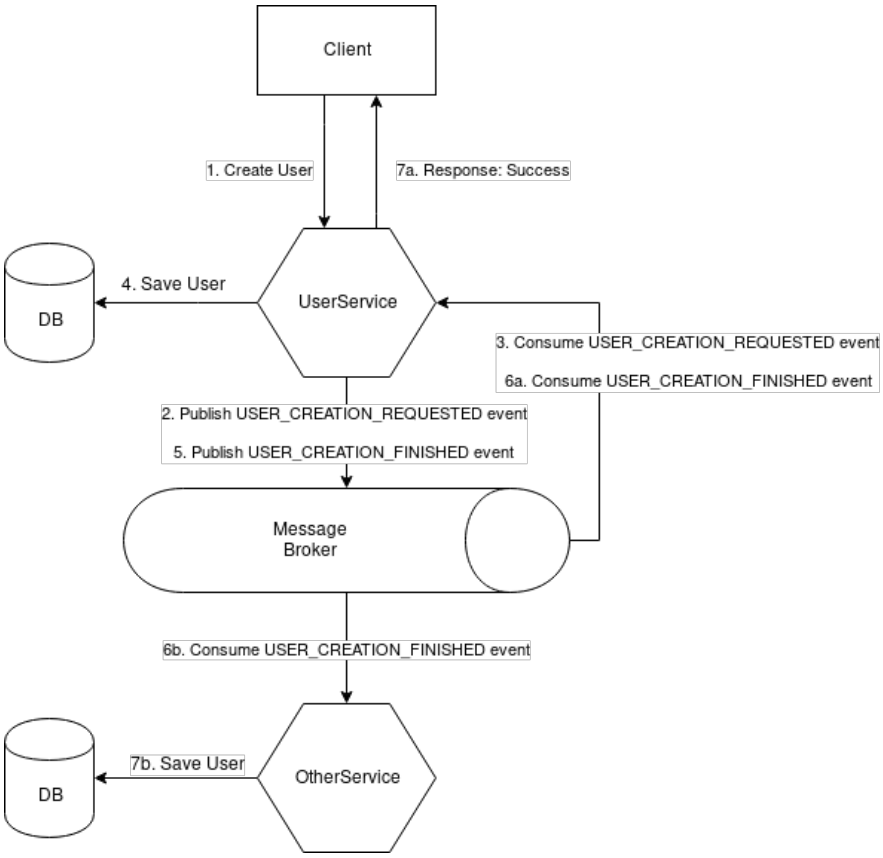


Figure 6.5: Implementation of the Listen to Yourself Pattern

The **Listen to Yourself Pattern** does not provide feedback to the user as stated in section 6.3.2. Due to the requirements in this context, the pattern was enhanced to settle this disadvantage. Figure 6.5 shows the result of this enhancement.

Instead of just using a UserCreatedEvent, the process was split into two steps. First, the UserCreationRequestedEvent is published. This event is received by the UserService which performs the database manipulation and publishes the UserCreationFinishedEvent. Note, that the database interaction does not necessarily happen on the same instance of the UserService that received the request from the client. The published UserCreationFinishedEvent can be used by other

services to trigger their actions, as well as by the UserService to send a response to the requesting client.

The UserCreationRequestedEvent has to be delivered once to the group of UserService instances in order to be performed once. The delivery of the UserCreationRequestedEvent reaches all service instances in order to reach the one that is in contact with the client in order to perform the response. The UserService stays operable during waiting on the UserCreationFinishedEvent by performing the waiting process in an asynchronous manner.

The application of this pattern brings a lot of complexity into the project. The mentioned mapping across threads from received events to open client requests that require the UserCreationFinishedEvent is error prone. Keeping track of all the different events is quite challenging. Every database manipulating operation can emit at least three different types of events: the request, the success and the failure of the operation. Additionally, there may be compensation events. In comparison to this very common scenario, creating a user, this seems to be an overkill. Especially with the goal to make the system easier to understand, the worth of applying this pattern is more than questionable.

The **Application Publishes Events Pattern** performed better in the implementation. Storing events happens alongside with executing the corresponding write-operation in a database transaction. The id of the events is generated by an auto-increment mechanism, which had to be implemented by hand because the underlying MongoDB does not support it out of the box.

Apart from this, the implementation progressed straight forward. Events are made accessible to users authenticated as admins by a read-only endpoint. The additional publishing application can poll this data and publish the new events. Comparing to the event-chaos of the Listen to Yourself Pattern, this implementation seems rather trivial. To fellow developers, it seemed far more reasonable to just store additional events instead of introducing a lot of complexity like mentioned above.

The fact, that the events are only stored and never removed can make event-sourcing possible. Newly created services that emerge into production can poll the whole history of events of other services and react to them. They can create their state as if they would have existed from the beginning because all state can be reproduced by making use of the events instead of the real data. [Fow17]

Given, that all services need a similar event publishing mechanism, standardizing the format of events may provide the benefit of reusing the publishing application for all services.

Without further analysis, it is obvious that the Application Publishes Events Pattern is the best choice in the context of the ODS.

7 Evaluation

This section describes the evaluation of this thesis. It reviews the requirements for the thesis defined in chapter 3 and discusses whether they are met. The upcoming sections each refer to the corresponding section of chapter 3.

7.1 Architecture Concept

The first and most important requirement was to create an architecture concept suited to the ODS.

Beginning with an analysis of the current monolithic implementation, a new architecture concept based on microservices was presented. Section 2.1 gives an overview of this new architecture and section 4.4.6 discusses an exemplary workflow. Appendix B documents a detailed description of all planned microservices which represents the result of this thesis. It shows that the whole application can be realized following the new architecture concept. Pitfalls that lead to a hidden monolith were identified and avoided.

An exemplary implementation in chapter 6 confirms as a proof of concept the feasibility of the developed architectural style. Because of that, this requirement is satisfied.

7.2 Migration Process

The second requirement was to design a migration process that can be applied to migrate the ODS from a monolithic architecture to microservices.

Section 5 introduces the proposed migration process on an abstract level, so it can be adapted and applied for almost any project that shall be migrated from a monolithic architecture to a microservice-based one. Specifically, section 5.2 presents the process of how to split a larger module into smaller services

which is an important part of the migration process. Section 5.3 introduces a list of important technologies. These are typically required in order to build an effective microservice-based architecture. They supplement the introduced migration process with valuable solutions to problems that can arise during the migration. These three sections together provide all the tools that are required to enable a seamless migration from a monolithic application towards microservices. Thus, the requirement of designing a migration process is fulfilled.

The process is applied in chapter 6 which validates the applicability as a proof of concept and fulfills the related requirement.

7.3 Implementation

The last requirement was to implement the developed architecture concept by example and evaluate the approach based on understandability, maintainability, and performance.

Based on the architecture description of section 4.4 the UserService was chosen as the candidate for the first microservice to be extracted. Section 6.1 shows the application of the extraction process.

An evaluation of different implementations of the distributed authentication regarding performance encourages the chosen approach to use microservices and a caching mechanism. Another refactoring step is described in section 6.2 regarding the technology choice. SpringBoot is a solid choice for microservices which enabled a fast implementation of the UserService. With other frameworks, better performance could be achieved, but the increased development experience was found to be more important for the project. The detailed measurements can be read up in section 6.2.2. Introducing a distributed architecture led to the risk of consistency issues. Section 6.3 presents two solutions and gives a clear recommendation to use the Application Publishes Events Pattern to make sure the consistency requirements of the ODS are met.

A detailed evaluation of how this architectural style affects the understandability, maintainability, and performance of the whole system is not possible at this point in time. Too many factors are unclear and the implementation by example in this thesis does not provide enough information to do so. If the microservice approach pays off in these regards can only be evaluated after further progress of the migration, but there are some indicators on the trends. Except for these limitations, the requirement regardings the implementation were met by pointing out the following trends regarding understandability, maintainability, and performance.

Dealing with distributed communication already introduced some complexity. But since this thesis established a suitable concept, the level of complexity should not rise significantly in this regard in the upcoming migration. Instead, it is visible that the separation of concerns by the microservices limits the scope of those and make them easier to understand and maintain. The responsibility of each is clearly defined and the focus on limited behavior reduces the complexity. By enabling to choose the suiting technologies for different use-cases, this modularization across services is advantageous compared to a modularization on a monolith. Summarizing, although some complexity had to be introduced to make the ODS suitable for a microservice-based architecture, the trend towards a system with better understandability and maintainability is presumable.

Regarding performance, there are trade-offs. The development team chose better development speed and experience over the performance of a running system. But this is surely also a kind of performance aspect that is important, especially in the early phases of a project. Additionally, the aspect of scalability must be considered to this point. The microservice approach promises to scale parts of the system that require additional resources very easily which affects the performance of the overall system as well.

7.4 Summary

Summarizing, the requirements regarding the architecture concept and the migration process were met. The implementation part of the thesis shows the feasibility of a microservice-based architecture, but there are limitations regarding the evaluation of understandability, maintainability, and performance.

Anyway, the implemented UserService combined with the migration process allows further migration of the ODS towards a microservice-based architecture, while being aware of the effects and pitfalls of a distributed architecture.

Appendix A ODS Vision Protocol 2018-12-19

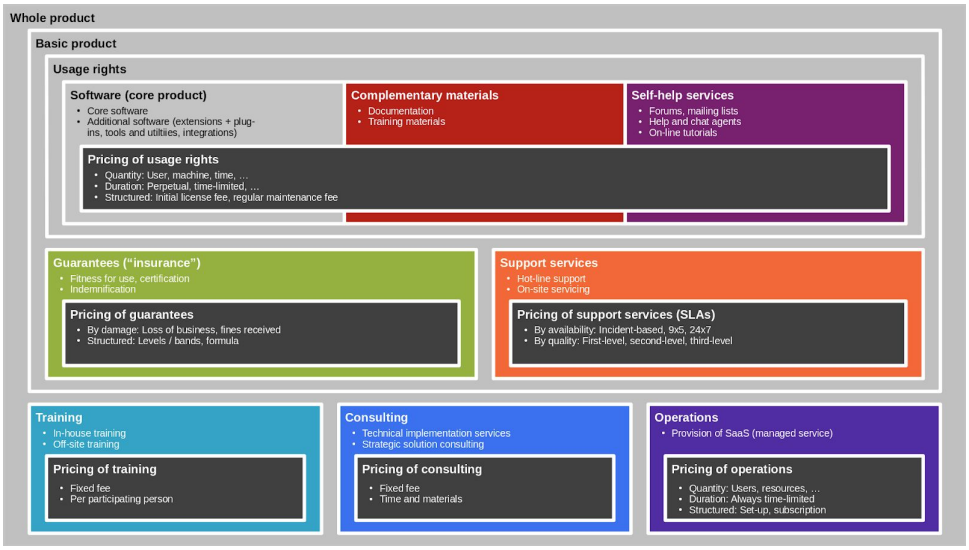
2018-12-19 ODS Vision Discussion

Andi, Georg, Mathias, Dirk

Meeting minutes

- Lean canvas / business vision
    - Problem
        - Finding interesting open data is hard
            - Govdata, searchability, etc. too hard
        - Using open data is (too) hard
            - Too much **effort**
                - Many different protocols
                - Many different data formats
            - Unsure **quality**
                - Unclear quality
            - Lack of **reliability**
                - Existing source are not reliable
        - Combining open data is (too) hard
            - Multiplies problem above
            - Legal situation potentially is unclear
                - Licenses and combination of licenses
                - Correlation and privacy
    - Solution
        - Makes adding easy
            - Clicky-bunti Configuration UI
        - Makes finding easy
            - Has broad meta data, easy search, all in one place (like a forge?)
            - Crowdsource makes it possible to scale, curate meta data
        - Makes using easy
            - Standardized way
                - One API, one query mechanism, etc. to access
            - Provides meta data about quality
            - Runs more reliable than original source
        - Makes combining easy
            - Help assess risk of license violation
            - Risk of privacy-violating correlation
    - Use-cases
        - Type of provision
            - Provision of software for open data to customers
                - Developed (channel) by consultancy or in-house developers
                - License sale / subscription model
            - Provision of service for open data to customers
                - Pure subscription for access
        - For which type customers
            - End-user through consultancy
                - In-house software application (both software and service)
            - End-user through software vendor
                - Variant 1: Random Android app (usually only service)
                - Variant 2: Non-app software product

        - Classic on premise software
    - Software vendor
        - Startup web service
        - Big data company
    - Data provider
        - For creating an ecosystem
        - For selling data
- By domain (we don't know yet)
    - Financial services
    - Automotive services
    - …
- Revenue sources



- Technical issues
    - Microservices breakdown
        - Database? MongoDB, Cassandra, Hadoop in a world of microservices
    - Configuration UI
        - Currently Javascript in sandbox
        - Future perhaps templates, something else
    - Query language
        - GraphQL? Unclear, perhaps multi-use of filter chains? Has nice existing UI ;-)
    - API backwards compatibility?
        - OK to break APIs and move faster, if we can adapt demo (marketing) apps later
- Research questions
    - Microservice related
        - Monolith to microservices migration patterns?
        - Group communication in microservices
        - RESTful design and bounded contexts
        - Independent deployment and shared libraries
        - Javascript (code-on-demand) and microservices
    - Open data (we still lack expertise)

        - Current providers of open data?
        - Who uses open data for what purpose?
        - What about data quality? Is that true? Analyse, quantify situation?
        - Data lineage? (Where did this data come from?)
        - Use of rules engine for quality improvement
        - ML for data quality improvement
        - Cross-validation using multiple data sources?
        - Resulting data license?
        - Data correlation valid?
    - Open data provision
        - Strategies for data storage, provision, archiving
- Further stuff
    - https://dirkriehle.com/2018/11/01/data-structures-vs-functions-in-the-age-of-microservices/

Appendix B Detailed Service Descriptions

B.1 UserService

description: „The UserService offers basic functionality for user management and authentication.“

endpoints:

GET /users
- descr: „Return List of all Users“
- resp: List<User>

POST /users
- descr: „Create new User“
- params:
  - body: User
- resp: User
- produces:
  - UserCreatedEvent
    - descr: „Event that user was created.“
    - routkey: user.created

GET /users/{id}
- descr: „Return information to one specific User.“
- params:
  - path {id}:
    - descr: „The user id.“
    - type: String
- resp: User

DELETE /users/{id}
- descr: „Delete one specific User.“
- params:
  - path {id}:
    - descr: „The user id.“
    - type: String
- produces:
  - UserDeletedEvent
    - descr: „Event that user was deleted.“
    - routkey: user.deleted

POST /users/{id}

descr: „Modify one specific User.“
params:
path {id}:
descr: „The user id.“
type: String
body: User
resp: User
produces:
UserUpdatedEvent
descr: „Event that user was updated.“
routkey: user.updated

GET /users/me

descr: „Return information to my (the logged in) User.“
resp: User

DELETE /users/me

descr: „Delete my (the logged in) User.“
produces:
UserDeletedEvent
descr: „Event that user was deleted.“
routkey: user.deleted

POST /users/me

descr: „Modify my (the logged in) User.“
params:
body: User
resp: User
produces:
UserUpdatedEvent
descr: „Event that user was updated.“
routkey: user.updated

B.2 DataSourceService

description: „The DataSourceService offers basic functionality for datasource and pipeline management.“

endpoints:

GET /datasources
- descr: „Return List of all DataSources“
- resp: List<DataSource>

POST /datasources
- descr: „Create new DataSource“
- params:
  - body: DataSource
- resp: DataSource
- produces:
  - DataSourceCreatedEvent
    - descr: „Event that datasource was created.“
    - routkey: datasource.created

GET /datasources/{id}
- descr: „Return information to one specific DataSource.“
- params:
  - path {id}:
    - descr: „The datasource id.“
    - type: String
- resp: DataSource

DELETE /datasources/{id}
- descr: „Delete one specific DataSource and all pipelines related to it.“
- params:
  - path {id}:
    - descr: „The datasource id.“
    - type: String
- produces:
  - DataSourceDeletedEvent
    - descr: „Event that datasource was deleted.“
    - routkey: datasource.deleted

GET /datasources/{datasourceId}/pipelines

descr: „Return information all pipelines related to one specific datasource.“

params:

path {datasourceId}:

descr: „The datasource id.“

type: String

resp: List<Pipeline>

POST /datasources/{datasourceId}/pipelines

descr: „Create new Pipeline“

params:

body: Pipeline

resp: Pipeline

produces:

PipelineCreatedEvent

descr: „Event that pipeline was created.“

routkey: pipeline.created

GET /datasources/{datasourceId}/pipelines/{id}

descr: „Return information to one specific Pipeline.“

params:

path {datasourceId}:

descr: „The datasource id.“

type: String

path {id}:

descr: „The pipeline id.“

type: String

resp: DataSource

DELETE /datasources/{datasourceId}/pipelines/{id}

descr: „Delete the specific pipeline.“

params:

path {datasourceId}:

descr: „The datasource id.“

type: String

path {id}:

descr: „The pieline id.“

type: String

produces:

PipelineDeletedEvent

descr: „Event that pipeline was deleted.“

routkey: pipeline.deleted

POST /datasources/{datasourceId}/pipelines/{id}
descr: „Modify the specific pipeline.“
params:
path {datasourceId}:
descr: „The datasource id.“
type: String
path {id}:
descr: „The pieline id.“
type: String
body: Pipeline
resp: Pipeline
produces:
PipelineUpdatedEvent
descr: „Event that pipeline was updated.“
routkey: pipeline.updated

B.3 PipelineSchedulingService

description: „The PipelineSchedulingService produces events that trigger the pipeline.“

consumes:

PipelineCreatedEvent
- descr: „Event that pipeline was created.“
- routkey: pipeline.created

PipelineUpdatedEvent
- descr: „Event that pipeline was updated.“
- routkey: pipeline.updated

PipelineDeletedEvent
- descr: „Event that pipeline was deleted.“
- routkey: pipeline.deleted

produces:

PipelineAdapterReadyEvent
- descr: „Event that pipeline is ready to be done. First job is an adapter..“
- routkey: pipeline.adapter-job.ready

B.4 AdapterService

description: „The AdapterService offers basic functionality for adapter documentation and execution.“

endpoints:

GET /adapters

descr: „Return List of all Adapters“
resp: List<Adapter>

GET /adapters/{id}

descr: „Return information to one specific Adapter.“
params:
path {id}:
descr: „The adapter id.“
type: String
resp: Adapter

consumes:

PipelineAdapterReadyEvent

descr: „Event that signals the next step of the pipeline is ready to be executed. The upcoming job is an adapter.“
routkey: pipeline.adapter-job.ready
produces:
PipelineTaskReadyEvent
descr: „Event that pipeline is ready to be done. Next job is a task.“
routkey: pipeline.task-job.ready

B.5 PipelineExecutionService

description: „The PipelineExecutionService offers basic functionality for executing pipeline jobs.“

consumes:

PipelineTaskReadyEvent

descr: „Event that the next step of a pipeline is ready to be executed. The upcoming job is a task.“

routkey: pipeline.task-job.ready

produces:

PipelineDataReadyEvent

descr: „Event that pipeline produced data.“

routkey: pipeline.data.ready

or PipelineNotificationReadyEvent

descr: „Event that pipeline produced a notification.“

routkey: pipeline.notification.ready

B.6 DataService

description: „The DataService offers basic functionality for data management.“

endpoints:

GET /data

descr: „Return List of all Data“
resp: List<Data>

GET /data/{id}

descr: „Return information to one specific Data.“
params:
path {id}:
descr: „The data id.“
type: String
resp: Data

consumes:

PipelineDataReadyEvent

descr: „Event that pipeline produced data. This data has to be inserted into the database.“
routkey: pipeline.data.ready

References

[All10] Subbu Allamaraju. *Restful web services cookbook: solutions for improving scalability and simplicity.* " O'Reilly Media, Inc.", 2010.

[Bal+18] Armin Balalaie et al. 'Microservices migration patterns'. In: *Software: Practice and Experience* 48.11 (2018), pp. 2019–2042.

[Con68] Melvin E Conway. 'How do committees invent'. In: *Datamation* 14.4 (1968), pp. 28–31.

[Fow05] Martin Fowler. 'ServiceOrientedAmbiguity'. In: *Martin Fowler-Bliki* (2005).

[Fow11] Martin Fowler. 'Cqrs'. In: *Martin Fowler's Blog* (2011).

[Fow17] Martin Fowler. 'What do you mean by „Event-Driven"?' In: *Martin Fowler's Blog* (2017).

[GL02] Seth Gilbert and Nancy Lynch. 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services'. In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[HM95] Martin Hitz and Behzad Montazeri. 'Measuring coupling and cohesion in object-oriented systems'. In: (1995).

[LF14] James Lewis and Martin Fowler. 'Microservices: a definition of this new architectural term'. In: *Mars* (2014).

[New15] Sam Newman. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[Ope14] Open Knowledge. *The Open Definition - Version 2.0.* 2014. URL: `http://opendefinition.org/` (visited on 10/05/2015).

[Pap08] Michael Papazoglou. *Web services: principles and technology.* Pearson Education, 2008.

[RGR17] David Reinsel, John Gantz and John Rydning. 'Data age 2025: The evolution of data to life-critical'. In: *Don't Focus on Big Data* (2017).

[Ric] Chris Richardson. *Pattern: Application publishes events*. https://microservices.io/patterns/data/application-events.html. Accessed: 2019-01-22.

[Sch] Hartmut Schlosser. *Technology trends 2018: Here are the top frameworks*. https://jaxenter.com/technology-trends-2018-frameworks-144575.html. Accessed: 2019-03-08.

[Sha17] Sourabh Sharma. *Mastering Microservices with Java 9: Build domain-driven microservice-based applications with Spring, Spring Cloud, and Angular*. Packt Publishing Ltd, 2017.

[Sho17] Oded Shopen. *Listen to Yourself: A Design Pattern for Event-Driven Microservices*. https://medium.com/@odedia/listen-to-yourself-design-pattern-for-event-driven-microservices-16f97e3ed066. Accessed: 2019-01-22. 2017.

[TL18] D. Taibi and V. Lenarduzzi. 'On the Definition of Microservice Bad Smells'. In: *IEEE Software* 35.3 (2018), pp. 56–62. ISSN: 0740-7459. DOI: 10.1109/MS.2018.2141031.

[Ver16] Vaughn Vernon. *Domain-driven design distilled*. Addison-Wesley Professional, 2016.

[Wol16] Eberhard Wolff. *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[Zin] Mathias Zinnen. *Design und Implementierung einer RESTful API fuer heterogene Daten*. https://osr.cs.fau.de/wp-content/uploads/2018/10/zinnen-2018-arbeit.pdf. Accessed: 2019-01-22.