Keep it in Sync! Consistency Approaches for Microservices An Insurance Case Study

Arne Koschel
Andreas Hausotter

Hochschule Hannover
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science
Hannover, Germany
Email: `arne.koschel@hs-hannover.de`
Email: `andreas.hausotter@hs-hannover.de`

Moritz Lange
Sina Gottwald

Hochschule Hannover
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science
Hannover, Germany
Email: `moritz.lange@stud.hs-hannover.de`
Email: `sina.gottwald@stud.hs-hannover.de`

***Abstract*—Microservices is an architectural style for complex application systems, promising some crucial benefits, e.g. better maintainability, flexible scalability, and fault tolerance. For this reason microservices has attracted attention in the software development departments of different industry sectors, such as e-commerce and streaming services. On the other hand, businesses have to face great challenges, which hamper the adoption of the architectural style. For instance, data are often persisted redundantly to provide fault tolerance. But the synchronization of those data for the sake of consistency is a major challenge. Our paper presents a case study from the insurance industry which focusses consistency issues when migrating a monolithic core application towards microservices. Based on the Domain Driven Design (DDD) methodology, we derive bounded contexts and a set of microservices assigned to these contexts. We discuss four different approaches to ensure consistency and propose a best practice to identify the most appropriate approach for a given scenario. Design and implementation details and compliance issues are presented as well.**

Keywords–Microservices; Consistency; Domain Driven Design (DDD); Insurance Industry.

I. Introduction

A current trend in software engineering is to divide software into lightweight, independently deployable components. Each component can be implemented using different technologies because they communicate over standardized network protocols. This approach to structure the system is known as the microservice architectural style [1].

As a study from 2019 (see [2]) shows, the microservice architecture style is already established in many industries such as e-commerce. However, this is not the case for the insurance and financial services industry. Therefore, as part of ongoing cooperation between the *Competence Center Information Technology and Management (CC_ITM)* and two regional insurance companies, the research project *Potential and Challenges of Microservices in the Insurance Industry* was carried out. The goal was to examine the suitability of microservice architectures for the insurance industry. The *CC_ITM* is an institute at the University of Applied Sciences and Arts Hanover. Main objective of the *CC_ITM* is the transfer of knowledge between university and industry. The cooperating insurance companies currently both operate a service-oriented architecture (SOA). Over time, however, it has become apparent that this architectural style is not suitable for some parts of the system and a finer subdivision (microservices) would be advantageous. A specific example for such a part of the system is the `Partner Management System`, which was transferred into a microservice architecture in the context of our research.

This paper presents a case study based on our research project. The case study focuses on consistency issues when implementing a microservice architecture. Therefore it describes how the monolithic architecture of the `Partner Management System` was divided into several microservices, which problems occur regarding the data consistency across the microservices and presents different approaches to solve these issues. Implementation details and insurance specific topics such as special compliance requirements are presented as well.

We organize the remainder of this article as follows: After discussing related work in Section II, we present the domain and requirements of the `Partner Management System` in Section III. Afterwards, Section IV shows how we split the system into microservices, discusses compliance aspects and describes the benefits the new architecture offers. Section V provides details about technical aspects of this architecture. In Section VI we evaluate the outcomes with a focus on consistency aspects. Section VII discusses general approaches to ensure consistency in microservice architectures and how these approaches can be applied to get a suitable consistency solution for the `Partner Management System`. Section VIII summarizes the results and draws a conclusion.

II. Related Work

Our research is based on the literature of well-known authors in the field of microservices, especially the ground works of Fowler and Lewis [1] as well as of Wolff [3]. For the practical parts of our research, mainly the elaborations of Newman (see [4]) were used. Moreover, we found valuable ideas (also) w.r.t consistency in the patterns work from Richardson [5]. Additional helpful microservices migration patterns are, for example, presented in [6] and some as well in [7]. Especially for the migration of the legacy application, the contribution of Knoche and Hasselbring (see [8]) was consulted. As a study from 2019 shows (see [2]), microservice architectures are barely found in the insurance and financial services industry in Germany. Therefore, results from other industries had to be used for our research (for example [9]).

Although the basic literature is extensive, not too much scientific research has been done about synchronizing services. Because microservices should use independent database schemes and can even differ in persistence technology, the traditional mechanisms of replicating databases (see, e.g., Tanenbaum and Van Steen [10, chap. 7]) cannot be applied as well. Instead, ideas and patterns from other areas of software engineering had to be transferred to the context of microservices.

So, in addition to general microservices research, more fundamental concepts of operating systems (e.g., [11]) and object-oriented programming (e.g., [12]) were considered by our research. Furthermore, ideas of general database research like the SAGA-Pattern [13], which was already applied to microservices by Chris Richardson [5], were considered as well. Event-based approaches like event sourcing, described by Fowler [14], were also applied to microservices within our research.

From a microservices design perspective, domain-driven design (DDD) from Evans [15], is currently considered to be a best practice to find suitable so called bounded contexts, which can form a functional basis for microservices. Going further than many microservices-based systems, we put a special emphasis on compliance aspects, when designing the bounded contexts. Since insurance companies are highly supervised by the government (in Germany: Federal Financial Supervisory Authority (BaFin)), several compliance rules apply for them, for example “Supervisory Requirements for IT in Insurance Undertakings (in German: VAIT [16])”.

Previous work has already evaluated the outcomes of the research project mentioned in the introduction (see [17]). However, the project and the evaluation only partially discussed the issues of consistency. In particular, it missed a fair bit of implementation details. Not discussed in [17] by us at all, are the aforementioned compliance aspects for our microservice design. For this reason, we have focused on those topics after the completion of the initial project, for example, with thesis work, that dealt with the issue of consistency in context of the project results as well a another work, which dealt with compliance aspects during service design.

In total, the present article is as such a significantly extended and updated version of our previous work, especially in the details for service boundaries, service compliance aspects, system architecture, and example implementation details [18].

III. Domain and Requirements of the Partner Management System

As mentioned in the introduction, one part of our research was migrating a system for managing partners of an insurance company - the `Partner Management System`. In this context, partners are defined as natural or legal persons who are in relation to the insurance company (e.g., clients, appraisers, lawyers or other insurance companies). Additional to personal information, a partner may also have information on communication, bank details, business relations and relations with other partners.

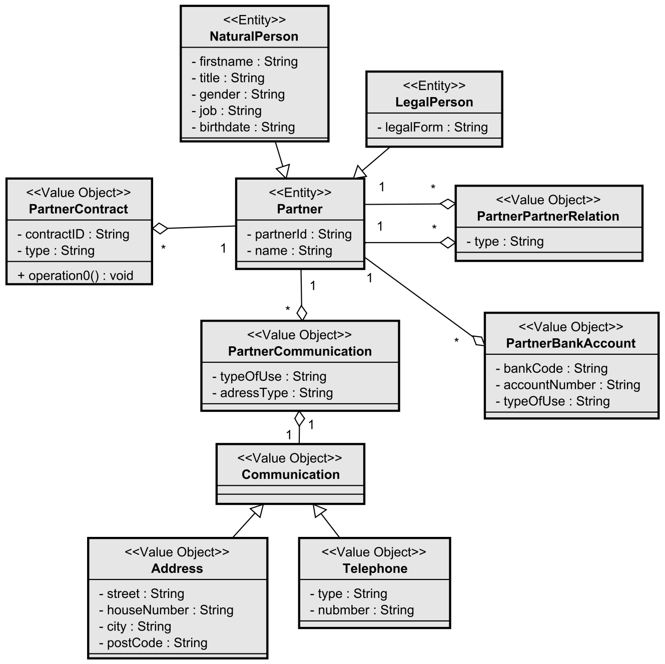


Figure 1. Simplified Model of the Overall Domain.

For introducing the overall domain, figure 1 shows a simplified model according to which the SOA service currently used by the partner companies was modelled. The presented model is strongly based on the reference architecture for German insurance companies (VAA) [19], which describes the monolithic way to implement the `Partner Management System`.

At its core, the system is a simple CRUD (Create, Read, Update, Delete) application that manages the entity `Partner` and its properties, even though the implementation as a single SOA service seems suitable at first glance.

Since the `Partner Management System` is a fundamental service of an insurance company, many other parts of the overall system are interested in the managed data. However, not all consumers of the `Partner Management System` are interested in the same subset of the data. Furthermore, some consumers should not have access to some data for security reasons. For example, the service that collects the monthly premiums does not need access to a client’s birth date or profession. These conditions result in a complex implementation of access rights and varying levels of stress for different parts of the `Partner Management System`. An efficient scaling is not possible. The `Partner Management System` is an atomic deployment unit that scales as a whole and fails as a whole.

Especially the inefficient scaling is critical in the case of partner companies, since the SOA service is implemented as a mainframe-based application that can only be scaled at great expense. Therefore, the partners use the low occupancy during the night to slowly persist all new datasets collected during the day so as not to overburden the mainframe-based application. However, in practice this approach makes crashes at night extremely critical as the entire system does not work all night and only few people are available to fix the problem.

The major issues of the current implementation of the `Partner Management System` are obviously a relatively poor flexibility, scalability and fault tolerance. In addition, the access rights management is so complex that a legally compliant implementation is difficult. This makes a microservices approach attractive for this use case.

IV. ANALYSIS OF THE MICROSERVICE ARCHITECTURE

In order to overcome the aforementioned limitations of the current `Partner Management System`, we designed a microservices-based approach. This section explains how the system has been split into independent services and presents the resulting benefits.

A. Dividing the Domain with DDD Strategic Design

In the context of our research, we used the principles of strategic design (see [15, Part IV]) as part of domain-driven design (DDD) to split the domain. Figure 2 shows the decomposition of the analysis model presented in the previous section.

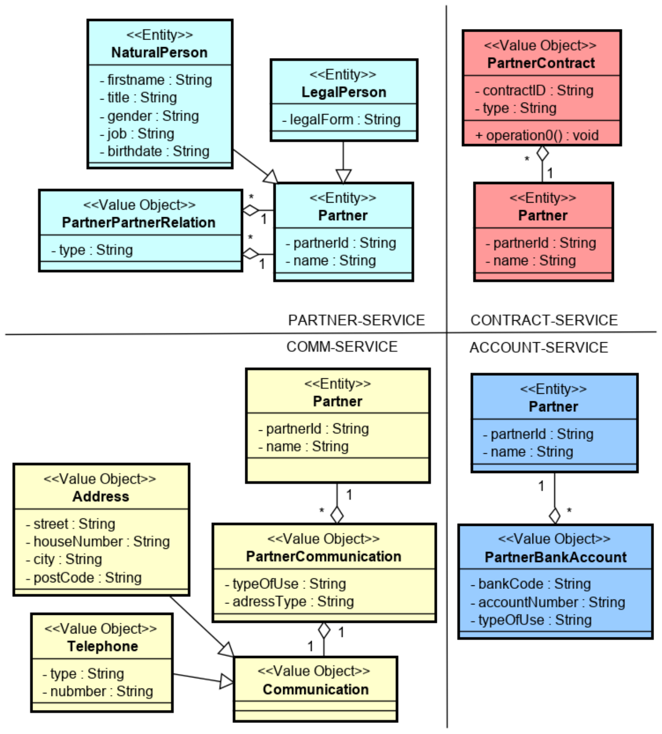


Figure 2. Bounded Contexts in the Microservice Architecture

It can be seen that the domain has been split into four bounded contexts. Bounded contexts are the central concept of strategic design. Vaughn Vernon (see [20]) describes it as an environment in which a specific language (ubiquitous language) is spoken and certain concepts are defined. Therfore, a bounded context is a conceptual boundary within which a particular domain model is applicable. In relation to Figure 2, this is shown by the fact that there are several separate models of the `Partner`, one for each bounded context. Every bounded context has a slightly different understanding of the entity `Partner`. For example, while for one context the `Partner` is a contractor, for another he is a bank account holder. In order to determine the presented contexts, we heavily discussed the domain and its processes with developers, architects and insurance domain experts. One technique that was used for this is event storming.

After we found the bounded contexts according to strategic design principles, the contexts were mapped to microservices. Leading microservices experts such as Chris Richardson [5] or Martin Fowler and James Lewis [1] recommend defining the microservices along the bounded contexts. This means that each bounded context should be implemented by one or more microservices. As figure 2 shows, in the case of the `Partner Management System`, a bijective mapping from bounded contexts to microservices was chosen. The system was divided into `partner-service`, `contract-service`, `comm-sevice` and `account-service`.

B. Compliance Aspects for Dividing the System

In addition to the better manageability of the business complexity as well as the more efficient scalability gained through the cut, the finer subdivision of the `Partner Management System` is also advantageous from a compliance point of view. This section describes the special requirements that insurance companies in Germany have to comply with and how these can be implemented by the designed microservice architecture.

TABLE I. Protection Level of Personal Data according to [21]

Protection Level	Personal data...	Example	Degree of Damage
A	...which have been made freely available by the persons concerned.	Data visible in the telephone book or on social media platforms.	minor
B	...whose improper handling is not expected to cause particular harm, but which has not been made freely accessible by the person concerned.	Restricted public files or social media not freely accessible.	minor
C	...whose improper handling could damage the person concerned in his social position or economic circumstances ("reputation").	Income, property tax, administrative offences.	manageable
D	...whose improper handling could significantly affect the social position or economic circumstances of the person concerned ("existence").	Prison sentences, criminal offences, employment evaluations, health data, seizures or social data.	substantial
E	...whose improper handling could impair the health, life or freedom of the person concerned.	Data on persons who may be victims of a criminal offence, information on witness protection program.	major

German insurance companies are supervised by the Federal Financial Supervisory Authority (BaFin) who published the "Supervisory Requirements for IT in Insurance Undertakings" [16]. These include instructions to comply with the basic principles of information security (confidentiality, integrity, availability) since insurance companies are considered as a critical infrastructure in Germany. This means that they are essential for society and economy and therefore more worthy of protection. Since 2018 the General Data Protection Regulation (GDPR) is enforceable for processing personal data as well. This also means that companies need to ensure data protection and privacy.

The microservice architecture divides the overall system into multiple independent components. Therefore, it allows to implement different protection levels for every microservice. This perfectly fits with the well known security engineering principle of compartmentalization [22]. In case of the `Partner Management System`, it allows us to meet the legal requirements according to [21], which are shown in Table I, more easily. It can be distinguished between more sensitive data requiring a higher security level like contract or bank details and data that is more uncritical like the name and telephone number of a person.

The microservice architecture of the `Partner Management System` only processes data belonging to protection levels A to C. Data like the address, telephone number and bank details of a person aren't as critical as, e.g., health data. Since the processed data in our system would cause less damage to the person affected if protection of the data would fail, there is no need to implement a protection level as strong as it should be for more critical data. But in general, due to the division in microservices we are able to map each service to the appropriate protection level.

V. Design and Implementation of the Microservice Architecture

After the previous section presented the functional design of the microservice architecture, this section shows the technical design and some implementation insights.

A. *The Technical Microservice Architecture*

Taking the bounded contexts found as input, the next step is the overall technical design of the microservice architecture.

Based on the technical specifications of the insurance companies involved, the resulting microservices are designed to be implemented as REST web services (see figure 3) in Java using the Spring framework. As mentioned before, each microservice should have its own data management, realized here as dedicated PostgreSQL databases. Instances of a microservice share a database (cluster). `partnerId` and `name` are kept in sync across all microservices using REST calls of the `partner-service`.

Parts of the Netflix OSS stack are used for the system infrastructure: Netflix Eureka as a service discovery and Netflix Zuul as an API gateway. Zuul also provides the web frontend of the application, which is realized as a single-page application using AngularJS. The ELK stack (Elasticsearch, Logstash and Kibana) is set up for monitoring and logging. All shown components of the architecture are deployed in separate Docker containers and connected by a virtual network using Docker Compose. In combination with the stateless architecture of the microservices, it is possible to run any number of instances of each microservice.

B. *Integrating Services into the Microservice Architecture*

Given the technical architecture of the overall microservices-based system, we now provide deeper technical details of particular microservices. Therefore, this section provides code snippets to show how services are integrated into the microservice architecture and how fault-tolerant calls between services are implemented.

```java
@EnableEurekaClient
@SpringBootApplication
public class Application {
 public static void main(String[] args) {
  SpringApplication
   .run(Application.class, args);
 }
}
```

Listing 1. Registration with Eureka Service Registry

To integrate a microservice into the microservice architecture from figure 3, it only needs to be registered with Eureka under a specific name. As a result, the API gateway (and any other microservice) can find instances of the microservice at runtime and use the provided REST endpoints. Fortunately, as listing 1 shows, Spring (Cloud Netflix) provides an easy way to register a microservice with Eureka.

As the listing shows, the class with the main method needs to be annotated with `@EnableEurekaClient`. In addition (not shown in the listing), the address of Eureka and the name under which the microservice should be registered must be stored in the `application.properties` file. As a result, the instance of the microservice now registers with Eureka when starting up. Note: In practice, there will usually be a federation of Eureka instances to ensure their availability.

C. *Implementing Network Calls*

Since network calls (to other microservices) can fail, it is useful to implement a fault tolerance mechanism. In our case,

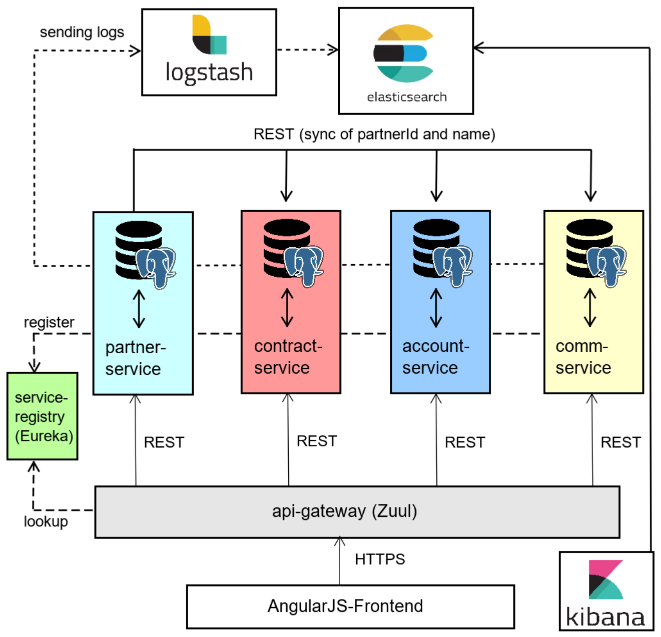


Figure 3. Technical Design of the Microservice Architecture

we used the Retry library from Spring. Just like the Eureka integration it is available via a simple annotation.

```
NaturalPerson createNP(NaturalPerson p){
 NaturalPerson savedInDB = repository.save(p);
 distribNP(p);
 log.info("A natural person was created.");
 return saved;
}

@Retryable(value={Exception.class}, maxAttempts=5)
void distribNP(NaturalPerson p) throws Exception{
 //network call
 dataDistributionService.update(savedInDB);
}

@Recover
void recover(Exception e, NaturalPerson p){
 //Rollback transaction to ensure
 //consistency and notify somebody
 //that something went terribly wrong.
}
```

Listing 2. Network Call with Retries

Listing 2 shows simplified parts of the implementation of the `partner-service`. Every time a new person is created, the change must be propagated to the other services via a synchronous network call. Since the successful execution of this call is a critical factor for consistency in the system, it makes sense to secure this call with a fault tolerance mechanism. The `@Retryable` annotation ensures that the annotated method is called several times if an exception occurs during execution. Listing 2 shows an implementation of five attempts to send the data to the other microservices. If an exception also occurs after the fifth time, the method annotated with `@Recover` is called instead.

D. Design of the microservices

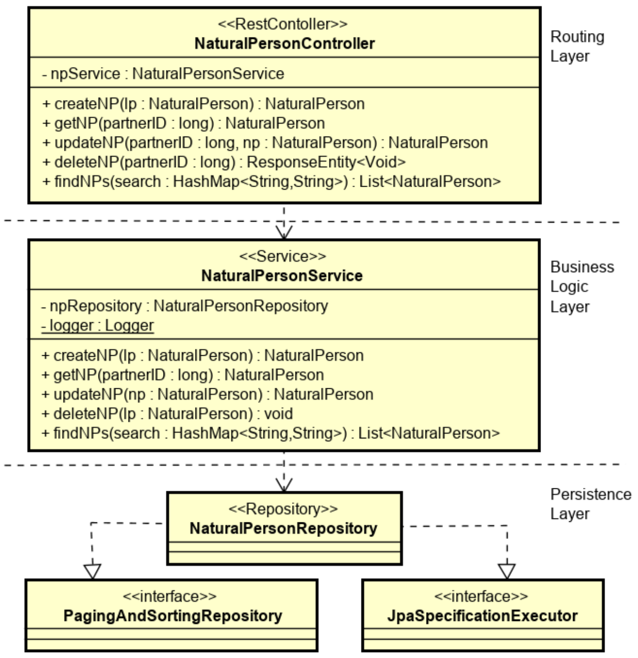
This section presents the internal architecture of the microservices. The services are implemented in Java using the Spring Framework. Parts of the `partner-service` serve as an example to show the general architecture of the services.

Figure 4 shows the architecture of the `partner-service`. It follows a layered architectural style. Each layer will be explained in the following.

E. Routing Layer

The routing layer is the one that interacts with the consumers of the service. With the annotation `@RestController` on the `NaturalPersonController`, the Spring Framework creates an instance of the class and delegates incoming HTTP requests to its methods. Each of the methods in the `NaturalPersonController` is responsible for a REST endpoint. The functional scope of the controller is limited to the CRUD operations and an interface to search for natural persons based on certain properties.

In addition to the `NaturalPersonController`, there is also a `LegalPersonController` and a `PartnerRelationsController` to provide the remaining functionality of the `partner-service`. Since the routing layer responds to incoming HTTP requests, it is responsible for encoding and decoding objects. In our



<<RestContoller>>
NaturalPersonController
- npService : NaturalPersonService
+ createNP(lp : NaturalPerson) : NaturalPerson
+ getNP(partnerID : long) : NaturalPerson
+ updateNP(partnerID : long, np : NaturalPerson) : NaturalPerson
+ deleteNP(partnerID : long) : ResponseEntity<Void>
+ findNPs(search : HashMap<String,String>) : List<NaturalPerson>
Routing Layer
<<Service>>
NaturalPersonService
- npRepository : NaturalPersonRepository
- logger : Logger
+ createNP(lp : NaturalPerson) : NaturalPerson
+ getNP(partnerID : long) : NaturalPerson
+ updateNP(np : NaturalPerson) : NaturalPerson
+ deleteNP(lp : NaturalPerson) : void
+ findNPs(search : HashMap<String,String>) : List<NaturalPerson>
Business Logic Layer
<<Repository>>
NaturalPersonRepository
Persistence Layer
<<interface>>
PagingAndSortingRepository
<<interface>>
JpaSpecificationExecutor

Figure 4. Architecture of the Partner-service

implementation, the objects are serialized as JSON. The layer is also responsible for catching exceptions from the business logic layer and translating them into HTTP status codes. In summary, the routing layer is an HTTP facade for the business logic.

F. Business Logic Layer

Classes in the business logic layer are provided with the annotation `@Service`. As a result, they are managed by the Spring Framework and can be used in other contexts using Spring's dependency injection mechanism. For example, the `NaturalPersonController` gets an instance of the `NaturalPersonService` injected to communicate with the business logic layer. As the name suggests, the business logic layer implements the business logic. In our implementation, the business logic is limited to some input validation and the communication with the persistence layer. Another responsibility of the layer is to log domain events. In our case, this is done with the SLF4J logging framework provided by Spring. In addition to the `NaturalPersonService`, there is also a `LegalPersonService` and a `PartnerRelationsService` to provide the remaining functionality of the `partner-service`.

G. Persistence Layer

The responsibility of the persistence layer is to commit run-time-objects to the database and convert persistent data to run-time-objects. Again, the dependency injection mechanism of Spring is used to give the layer above access to the functionality. The persistence layer is implemented using Spring Data JPA and is therefore limited to the definition of an interface with the annotation `@Repository`. Managed entity classes must be annotated with `@Entity`. For the search functionality described above and the possibility to sort search results, the repository is extended with another two interfaces from Spring Data JPA. For clarity, the methods of the persistence layer are not shown in figure 4. The exact description of Spring Data JPA can be found in the documentation of the Spring Framework. In addition to the `NaturalPersonRepository`, there is also a `LegalPersonRepository` and a `PartnerRelationsRepository`. Each repository is responsible for a specific database table.

VI. Challenges of the microservice architecture

Looking at the architecture described in section IV, it looks like the microservice architecture can solve the problems of the currently implemented monolithic system. In particular, the scalability and fault tolerance of individual parts of the system are a crucial advantage compared to the current implementation. The system can adapt to the changing load during the day, eliminating the need for risky nightly batch jobs. With the benefits of finer granularity however, there are also many new challenges that need to be mastered. One major challenge, for example, is the distributed monitoring and logging, which is handled by the ELK stack. As already mentioned, another key challenge is the consistency assurance across the services. This means the synchronization of the `partnerId` and `name`, which serves as an example for the application of our research results.

As mentioned briefly in Section IV, the synchronization is realized by REST calls of the `partner-service`. Whenever a `partnerId` or `name` record is created, deleted or changed, the `partner-service` distributes this information to the other services in a synchronous way. This means that the `partner-service` is responsible for ensuring the consistency of the overall system. Furthermore, the development team of the `partner-service` is responsible for not corrupting the other contexts by changing the data model of `Partner`. This approach is known as Customer/Supplier pattern (described by Evans [15]) in the context of DDD. If it is likely that the data model changes, an anticorruption layer should be considered.

Even if, e.g., the `partner-service` is unavailable, the other services can still resolve foreign key relationships to partner data, because they keep a redundant copy. Moreover, the system reduces service-to-service calls, because other services don't need to call the `partner-service` on every operation. This ensures loose coupling of services, which is a key aspect of microservice architectures [1].

The synchronization of `partnerId` and `name` is a critical part of the application, which is why its implementation needs to be closer discussed. Since the goal of the first phase of the project was building the architecture in general, a synchronous solution was chosen for simplicity. This has several drawbacks:

- **Fault tolerance.** If the `partner-service` crashes during synchronization, some services might not be notified about the changes. Conversely, if another service is not available for the `partner-service`, it will not be notified as well. This is due to the transient characteristic of REST calls.
- **Synchronicity.** After a change of `partnerId` or `name`, a thread of the `partner-service` is in a blocked state until all other services have been notified. Because multiple network calls are necessary for the synchronization, the response time of the `partner-service` is affected. Since microservices should be lightweight, a large number of network calls and busy threads are a serious problem.
- **Extensibility.** Extensibility is a key benefit of the microservices approach. In the current implementation, the `partner-service` holds a static list of services that need to be notified upon a change of `partnerId` or `name`. If a new service is added to the system, which is interested in partner data, the `partner-service` must be redeployed. Additionally, the bigger the number of services to notify gets, the more the response time of the `partner-service` is impaired.

As part of our research, further alternative solutions were explored, which will be discussed in the next sections.

VII. Consistency Assurance in the Partner Management System

In order to find a suitable solution for the specific problem of synchronizing `partnerId` and `name`, the general approaches have to be examined.

A. General approaches

The central research question is how a change of master data can be propagated to other interested services without breaking general microservices patterns like loose coupling and decentral data management. Especially the latter makes this a major challenge: Because the data stores and schemas should be separated, the standard mechanisms of synchronizing databases cannot be used here.

Based on our research, there are four possible solutions for synchronizing redundant data in microservices:

- **Synchronous Distribution.** One approach is that the owner of the data distributes every change to all interested services. As discussed in section VI, our microservice architecture already follows this approach. To provide loose coupling however, the addresses of services to be notified should not be contained in the master service's code. A better solution is to hold those addresses in configuration files, or even better, establish a standard interface where services can register themselves at runtime. For example, an existing service registry (e.g., Netflix Eureka) can be used to store the information which service is interested in which data. This approach roughly corresponds to the Observer-Pattern of object-oriented software development, where the subject registers at the observer to get synchronously notified when changes are made.
  As already discussed, notifying a large number of interested services might cause significant load of the service containing the master data. This can become a disadvantage. The distribution takes place in a synchronous fashion however, directly after the change of data itself. This means that this solution provides a high degree of consistency among services as long as the requests don't fail.
- **Polling.** Another solution is to relocate the responsibility of synchronizing the redundant data to the interested services themselves. A straightforward approach

is to periodically ask for new data using an interface provided by the service containing the master data. Based on timestamps, multiple data updates can be transferred in one go. The size of the inconsistency window can be controlled by each interested service independently via the length of the polling interval. However, despite being consistent in the end, the time frame in which the data sets might differ is a lot larger than the one when using a synchronous solution. This model of consistency is known as eventual consistency (see [23]).

- **Publish-Subscribe.** To completely decouple the service containing the master data from the other services, a message queuing approach can be utilized. On every data change, an event is broadcasted on a messaging topic following the Publish-Subscribe-Pattern. Interested services subscribe to this topic, receive events and update their own data accordingly. Multiple topics might be established for different entities. If the messaging system is persistent, it even makes the architecture robust against service failures. This approach is suitable for the resilient and lightweight nature of microservices. It must be noted, however, that it also falls in the category of eventual consistent solutions - until the message is delivered and processed, the system is in an inconsistent state. For example, the SAGA-Pattern uses this approach to distribute information about changes.
- **Event Sourcing.** Instead of storing the current application state, for some use cases it might be beneficial to store all state transitions and accumulate those to the current state when needed. This approach can also be used to solve the problem of distributing data changes. Upon changes in master data, events are published. Unlike the Publish-Subscribe solution however, the history of all events is persisted in a central, append-only event storage. All services can access it and even generate their own local databases from it, each fitting their respective bounded context. This solution provides a high degree of consistency: Each data change can be seen immediately by all other components of the system. It must be noted that a central data store, which microservices try to avoid, is introduced. This weakens the loose coupling and might be a scalability issue - the append-only nature of the data storage enables high performance though.

If none of the consistency trade-offs above is bearable, this might be an indicator that the determined bounded contexts are not optimal. In some cases, contexts are coupled so tightly that keeping data redundantly is not feasible. In this case, it should be discussed if the contexts and therefore also services can be merged. Furthermore, if this issue occurs in several parts of the architecture, it should be evaluated whether a microservices approach is the right choice for this domain.

B. Best practice for synchronizing partner data

The discussion in the previous section has shown that some approaches tend to guarantee a stronger level of consistency than others. This means that before all non-functional requirements can be considered as decision criteria, the required consistency degree of the underlying business processes must be examined. This can be done by first specifying the possible inconsistent states and then combining them with typical use cases of the system.

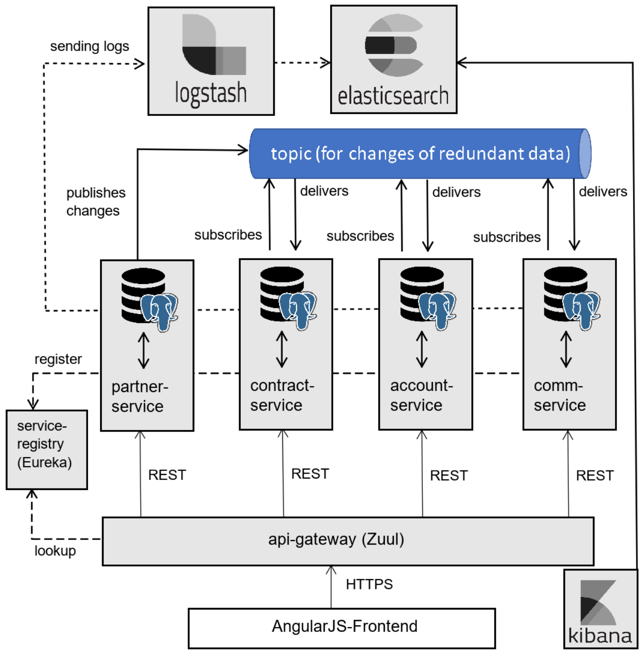


Figure 5. `Partner Management System`: Publish Subscribe Solution

In case of the `Partner Management System`, only the partner data, containing the `partnerId` and `name` of every partner, is saved in a redundant fashion. Combining these with the CRUD-operations, the following inconsistent states are possible:

- A new `Partner` might not yet be present in the whole system.
- `partnerId` or `name` might not be up-to-date.
- A deleted `Partner` might not yet be deleted everywhere.

We combined these inconsistent states with typical use cases and business processes in which the `Partner Management System` is involved, like sending a letter via mail or a conclusion of an insurance contract.

The result of this examination is that the partner management of insurance companies is surprisingly robust against inconsistent states. This is mainly due to the reason that the business processes itself are already subject to inconsistency: If a customer changes his or her name, for example, the inconsistency window of the real world is much larger than the technical one (the customer, e. g., might not notify the insurance company until several days have passed). The postal service or bank already needs to cope with the fact that the name might be inconsistent. The discussion of other potential situations brought similar results. This makes sense because the business processes of insurance companies originated in a time without IT, which means that they are already designed resilient against delays and errors caused by humans. Cases where the customer notices the delay (e.g., a wrong name on a letter) are rare and justifiable.

The combination of the inconsistent states and the use cases of the `Partner Management System` revealed that a solution which promotes a weaker consistency model is acceptable – no approach has to be excluded beforehand. So, the choice of a synchronization model is only influenced by the non-functional requirements.

The analysis of the partner domain showed that the main non-functional requirements are loose coupling, high scalability and easy monitoring. Especially because of loose coupling, the Publish-Subscribe-Pattern is the most viable solution.

Figure 5 shows how the Publish-Subscribe solution can be used to synchronize `partnerId` and `name`. On every change of the partner data, an event is broadcasted by the `partner-service`. The other services subscribe to this topic and receive those events and update their own data accordingly. Technically, the topic could be implemented by, for example, RabbitMQ and a standardized messaging protocol like AMQP.

VIII. Conclusion and Future Work

This paper has presented a microservice case study from the insurance industry. The challenges of the existing `Partner Management System` were identified and discussed. The paper gave insights into the design process of the new microservice architecture, presented implementation details and discussed further (insurance-specific) topics such as special compliance requirements. In addition, the new challenges arising from the microservice architecture were discussed and the implementation, which was initially developed during our research project, was critically reviewed with regard to consistency. The general solutions for the synchronization of data in distributed systems were pointed out and a (more suitable) alternative to the current implementation was determined.

The next steps will be to complete the implementation of the publish-subscribe approach for the `Partner Management System` with the described technologies. Prior to take the system into operation, it must be comprehensively tested under real-world conditions. In our recent research a test strategy tailored to the requirements of the industry partner was designed and implemented. This strategy comprises several steps, i.e. interface tests, web interface tests, load and performance tests, which are performed within so-called 'layers'. These layers represent different test environments, up to an infrastructure which is similar to the production environment. To exploit the potential of the strategy it is reasonable to integrate the tests into a CI/CD (continuous integration / continuous delivery) pipeline as part of the pipeline's test stage. After completing the implementation of the `Partner Management System`, the tests of the system will be performed based on the test strategy.

To demonstrate the approach for finding a suitable consistency assurance solution, the example of the `Partner Management System` is sufficient. To further underpin our findings, however, they need to be applied to more complex examples.

References

[1] M. Fowler and J. Lewis, "Microservices a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, March 2014, [retrieved: 05, 2020].

[2] H. Knoche and W. Hasselbring, "Drivers and barriers for microservice adoption–a survey among professionals in germany," Enterprise Modelling and Information Systems Architectures (EMISAJ), vol. 14, 2019, p. 10.

[3] E. Wolff, Microservices: Flexible Software Architecture. Addison-Wesley Professional, 2016.

[4] S. Newman, Building microservices: designing fine-grained systems. O'Reilly Media, Inc., 2015.

[5] C. Richardson, Microservices Patterns: With examples in Java. Manning Publications, 2018.

[6] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," Software: Practice and Experience, vol. 48, no. 11, 2018, pp. 2019–2042.

[7] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," IEEE Software, vol. 33, no. 3, 2016, pp. 42–52.

[8] H. Knoche and W. Hasselbring, "Using microservices for legacy software modernization," IEEE Software, vol. 35, no. 3, 2018, pp. 44–49.

[9] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, 2017, pp. 243–246.

[10] A. S. Tanenbaum and M. Van Steen, Distributed Systems: Pearson New International Edition - Principles and Paradigms. Harlow: Pearson Education Limited, 2013.

[11] A. S. Tanenbaum, Modern Operating Systems. New Jersey: Pearson Prentice Hall, 2009.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software. Amsterdam: Pearson Education, 1994.

[13] H. Garcia-Molina and K. Salem, "Sagas," vol. 16, no. 3. ACM, 1987.

[14] M. Fowler, "Event Sourcing," https://martinfowler.com/eaaDev/EventSourcing.html, December 2005, [retrieved: 05, 2020].

[15] E. J. Evans, Domain-driven Design - Tackling Complexity in the Heart of Software. Boston: Addison-Wesley Professional, 2004.

[16] "Versicherungsaufsichtliche Anforderungen an die IT (VAIT) (2019) vom 20.03.2019," https://www.bafin.de/SharedDocs/Downloads/DE/Rundschreiben/dl_rs_1810_vait_va.html, March 2019, [retrieved: 05, 2020].

[17] M. Lange, A. Hausotter, and A. Koschel, "Microservices in Higher Education - Migrating a Legacy Insurance Core Application," in 2nd International Conference on Microservices (Microservices 2019), Dortmund, Germany, 2019, https://www.conf-micro.services/2019/papers/Microservices_2019_paper_8.pdf, [retrieved: 05, 2020].

[18] A. Koschel, A. Hausotter, M. Lange, and P. Howeihe, "Consistency for Microservices - A Legacy Insurance Core Application Migration Example," in SERVICE COMPUTATION 2019, The Eleventh International Conference on Advanced Service Computing, Venice, Italy, 2019, https://www.thinkmind.org/index.php?view=article&articleid=service_computation_2019_1_10_18001, [retrieved: 05, 2020].

[19] GDV, "The application architecture of the insurance industry – applications and principles," 1999.

[20] V. Vernon, Implementing domain-driven design. Addison-Wesley, 2013.

[21] "Schutzstufenkonzept des LfD Niedersachsen," https://www.lfd.niedersachsen.de/technik_und_organisation/schutzstufen/schutzstufen-56140.html, October 2018, [retrieved: 05, 2020].

[22] A. Roland, "Secrecy, technology, and war: Greek fire and the defense of byzantium, 678-1204," Technology and Culture, vol. 33, no. 4, 1992, pp. 655–679.

[23] W. Vogels, "Eventually Consistent," Commun. ACM, vol. 52, no. 1, Jan. 2009, pp. 40–44. [Online]. Available: http://doi.acm.org/10.1145/1435417.1435432, [retrieved: 05, 2020]