An Experience Report from the Migration of Legacy Software Systems to Microservice based Architecture

***Abstract*—*Context:* The literature provides evidence of challenges and difficulties related to the migration of legacy software systems to a microservice based architecture. The idea of microservices stipulates that the software be organized as a suite of small, modular, and independently deployed services, in which each service runs on its own process and communicates through well-defined, lightweight mechanisms to serve a business goal. However, the literature lacks step-by-step guidelines telling practitioners how to accomplish the migration from an existing, monolithic structure to a microservice based architecture. *Goal:* Discuss lessons learned from the migration of legacy software systems to microservices-based architecture. *Method:* We conducted two studies (a pilot and a case study) aiming at characterizing the relevants steps of such guidelines. *Results:* We report the steps and challenges observed during the migration reported in this study. *Conclusion:* We identify at least three main phases that drive the migration process.**

***Index Terms*—monolithic legacy systems, exploratory study, microservices**

I. Introduction

Microservices relate to an architectural style inspired by service-oriented computing [1] and comprise a promising solution to efficiently build and manage complex software systems [2]. Adoption of a microservices-based architecture promises to obtain cost reduction, quality improvement, agility, and decreased time to market. Microservices can be approached as the software equivalent of *Lego* bricks: after they are proven to work they fit together appropriately. They are an option to construct complex solutions in less time than with traditional architectures [2].

Many legacy software systems moved to the cloud without prior adjustments in their architecture for the new infrastructure. Many of them have been originally placed in virtual machines and deployed in the cloud, assuming the characteristics of resources and services of a traditional data center. This approach fails to reduce costs, improve performance and maintainability [3].

A open question concerns the steps that should be followed to migrate a monolithic legacy system to a microservices-based architecture. To the best of our knowledge, just a few works discussed this issue [4]–[6]. To fill this gap, we present the lessons learned when migrating two legacy systems, which were acquired in a two-phase study. It addresses the following Research Question (RQ): *Which steps should be performed to support the migration of legacy software systems to microservices-based architecture?* Lessons learned helping practitioners from industry and academia in migrating legacy systems to microservices can contribute in encourage the embracing of this challenge.

The rest of this paper is organized as follows. Section II discusses the main shortcomings of a monolithic legacy system and map them to possible solutions provided by the microservice-based architecture. In Section III, we report a two-phase study in which the first phase is a *pilot study* aimed at identifying key steps of the migration process as well as improvement opportunities. The second phase is presented in Section IV and comprises a *case study* to apply a reviewed and improved version of the steps performed in the first study. Section V reports lessons learned and discusses tasks related to the migration based on experience acquired. Section VI discusses opportunities for future research and provides concluding remarks.

II. Monolithic vs Microservices

The use of single executable artefacts or *monoliths* and the modularization of their abstractions, rely on the sharing of resources of the same machine (memory, databases and files) [1]. Since the parts of a monolithic system depend on shared resources, they are not independently executable [1], [7], [8]. Large monolithic systems are difficult to maintain and evolve due to their complexity [1]. Tracking down bugs in these conditions requires much effort and is thus likely to diminish team productivity [1]. Moreover, adding or updating libraries risks producing inconsistent systems that either do not compile/run or worse, *misbehave* [1]. A change performed on a monolithic system entails the re-building of the whole application. As the system evolves, it becomes ever more difficult to mantain it and keep track of its original architecture. This can result in recurring downtimes, specially for large projects, hindering development, testing, and maintenance activities [1]. Monolithic systems under these conditions are prone to stop working and become unable to provide part or all of their functionality. They also suffer from scalability issues. To deal with the shortcomings of this type of applications and to handle an unbound number of requests, developers create new instances of them and split the load among these instances. Unfortunately, this approach is not effective, since the increased traffic will be targeted only to a subset of the modules, causing difficulties for the allocation of new resources for other components [1]. Microservices should be small and independent enough to allow the rapid development, (un)pluggability, independent evolution and harmonious coex-

istence. Microservices have been referred as a solution to most of the shortcomings of monolithic architecture. They use small services to remove and deploy parts of the system, enable the use of different frameworks and tookits and to attain increased scalability and better overall system resilience. A microservice architecture can make use of the elasticity and better pricing model of cloud environments [9].

Next follows a non-exhaustive list of advantages that stand out when using microservices: cohesive and loosely coupled services [10]; independent implementation of each microservice and thus adaptability [11]; independence of multifunctional, autonomous and organized teams to provide commercial value, not just technical characteristics [11]; independence of domain concepts [10]; freedom from potential side effects (SPoF) in services; encourage of the *DevOps* culture [9], which basically represents the idea of decentralizing skills concentration into multifunctional teams, emphasizing collaboration between developers and teams, ensuring reduced lead time and greater agility during software development.

III. A Two-phase Study

This section describes the design and settings of a two-phase exploratory study with the goal of identifying relevant and effective steps of a migration of legacy systems to a microservices-based architecture. Exploratory studies are intended to lay the groundwork for further empirical work [12]. The study aims to address the following RQ: *What would be the set of effective steps to migrate legacy systems to a microservices-based architecture?* The specific research questions (SRQ's) derived from the base RQ are as follows: *SRQ1: How to find features in a legacy application so that they can be subsequently modularized and become a candidate to a microservice-based architecture?* and *SRQ2: How to migrate the best candidate features to a microservice-based architecture?*

The Study Protocol. The first author of this paper carried out the tasks of the reported study, after discussing the strategies, experiences and impressions with the other two authors. To answer the research questions (primary and secondaries), all steps registered by the first author in manuscripts were analyzed. ***The Legacy Systems***. Candidate applications for this study should match the following characteristics: (1) be a legacy application, (2) have a monolithic architecture that does not have its functionalities modularized, (3) show symptoms of scattering and tangling, and (4) structurally correspond to the *Big Ball of Mud* anti-pattern [13]. ***Expected Outcome.*** In contrast, we expect the evolved version of the application to be be more coherent, loosely coupled, showing a modular decomposition more aligned to the services it provides [14]. We also expect to witness an increase in the autonomy of developing teams within the organization, as new functionalities can be localized within specific services [14].

DDD Key Concepts. To accomplish the tasks of this study, we used key Domain-Driven Design (DDD) concepts to translate functionalities into domain and subdomain and thereby support the migration. A *Bounded Context* is a subsystem

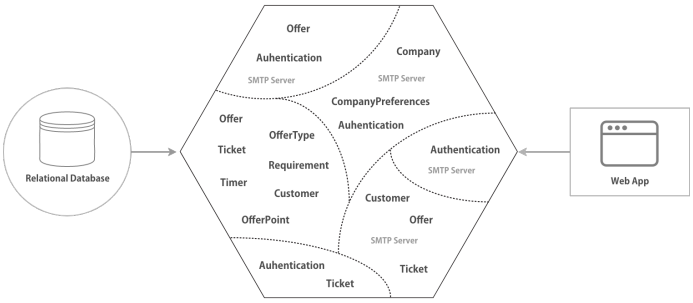

Fig. 1. Entities and Associated Features Scattered and Tangled in ePromo

in the solution space with clear boundaries that distinguish it from other subsystems [15]. *Bounded Context* aids in the separation of contexts to understand and address complexities based on business intentions. The *Domain* in the broad sense is all the knowledge around the problem one is trying to solve. Therefore, it can refer to either the entire *Business Domain*, or just a basic or support area. In a *Domain*, we try to turn a technical concept with a model (*Domain Model*) into something understandable. The *Domain Model* is the organized and structured knowledge of the problem. This model should represent the vocabulary and main concepts of the domain problem and identify the relationships between all entities. It should act as a communication tool for all involved, creating a very important concept in DDD, which is *Ubiquitous Language*. This model could be a diagram, code examples or even written documentation of the problem. The important thing is that the *Domain Model* must be accessible and understandable by all involved in the project.

A. The Pilot Study

The *ePromo* system was selected as the subject of the Pilot study. It comprises a typical example of a corporate/business coupon web system implemented in the `PHP` programming language for the management of outreach campaigns. The web server is *Nginx* and its features include: creation of personalized offers and issuance of tickets made by the customer. All functionalities are implemented in a large artifact, connected to a single relational database (*MySQL*), whereas *Memcached* is used as a memory cache system, including data related to the sessions - signs of a monolithic application. Due to the sudden growth of demand for coupons, the application started to face problems in this specific component, which led to interruptions in the system operation.

In order to answer *SRQ1 (find features to be subsequently modularized and turned into microservice candidates)*, the participant applied a manual identification of candidate features and their respective relationships, by navigating among the directories and files and identifying the purposes of each class. Figure 1 illustrates the identified entities were: `Offer`, `OfferPoint`, `Ticket`, `Requirement`, `Timer`, `User`, `Company` in the beginning of the pilot study. By analysing the features associated to these entities, we acquired an initial perception of how they are tangled and scattered in the code. In

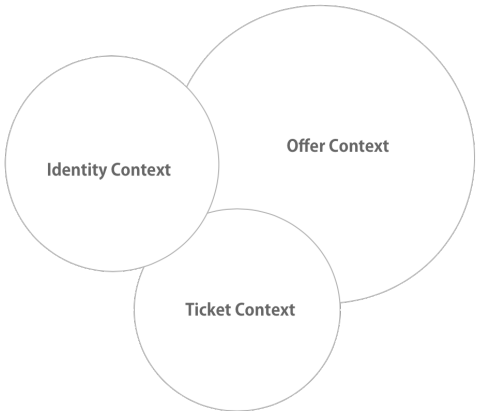

Fig. 2. ePromo System Context Map in the Pilot Study

fact, the functionalities are the reference to build the context map. It is worth mentioning that during the elaboration of the context map based on information retrieved from the source code, it was possible to recognize the entities and the candidates for value object's and aggregate roots. At this time, we had the opportunity to spot code tightly coupled to the web framework, right at the initial browsing stage.

We decided to deal with one feature at a time, based on the list of features. We started with the functionality that would have lowest impact when compared to the others. This would enable the validation of boundaries between features with the least risk of side effects. Considering that the business rules were scattered throughout the controllers with significant duplication, additional effort to identify the various functionalities involved was required. This scenario also indicated a symptom of tangling.

When analyzing the `TicketsController` artifact it was noticeable that it has many responsibilities and that its business rules were scattered. It needed extensive refactoring, including extraction of clear layers for different levels of abstraction. Each layer would be represented by a folder, which entails structural changes at that level, within the repository's source root.

New directories have been created: `Application`, `Domain` and `Infrastructure`. Folder `Application` is to be devoid of business logic and be responsible for connecting the user interface to the lower layers, i.e., the application layer will be able to communicate with the domain layer, which will act as a sort of public API for the application. It will accept requests from the outside world and return answers appropriately. Folder `Domain` is to harbour all concepts, rules and business logic of the application, such as the user entity or the user repository. These files will be stored according to the context identified in previous steps. Folder `Infrastructure` is to host the implementations concerning technical features, which provide support to the layers above, namely persistence, and communication over networks.

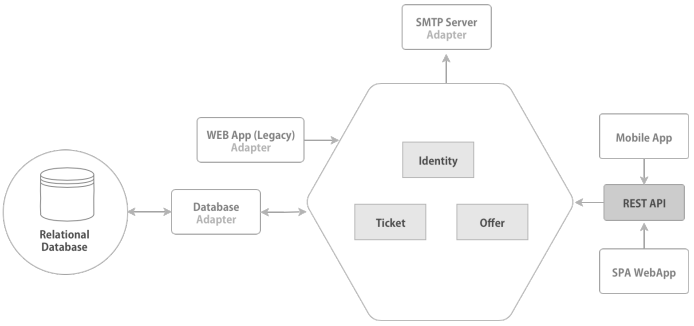

Fig. 3. ePromo Modularized Version (End of the Pilot Study)

At this point, we applied the *Command* pattern [16] to minimize coupling and deal with the tangled code with scattered business rules and identified in the controllers of the application. *Command* encapsulates a request as an object, thereby parametrizing clients with different requests, queue or log requests, and support undoable operations [16]. Based on `TicketController`, *Command* was used to uncouple the controller from the user interface logic. When looking at the commands, we should be able to spot the goal of that code snippet. The controller is to pass just the information needed by the command - `CreatingTicket` in this case - to forward to the handler, which will handle the acceptance of the command and will complete its task. This approach brings several advantages, namely: (1) the functionality can run in any part of the application; (2) the controller will no longer have business rules, doing just what is proposed above; (3) as a result of decoupling, the tests can be made easier. The new version of the modularized system is presented in Figure 3.

B. Lessons Learned from the Pilot Study

Based on the experience gained in the pilot study, we can answer *SRQ1* as explained in the following. In Section III-A, we described that the identification of functionalities faced difficulties due to the existence of lots of classes with repeated business rules and scattered throughout. This situation is typified as the *Anemic Model* anti-pattern[1]. Therefore, identifying business resources requires much effort. During the identification and mapping of the business contexts, we noticed that despite the sudden growth of demand for coupons, the number of features candidates for microservices may not be indicative of the use of a microservice architecture. There is not a positive trade-off between the advantages of microservices and the corresponding costs and effort required to manage it [2]. Although microservices approaches offer substantial benefits, the corresponding architecture requires extra machinery, which may also impose substantial costs [2]. This would will give rise to greater complexity, which is incompatible with the relative simple scenario now perceived through the map of contexts. Therefore, the decision for the migration should bear in mind the extra effort required to work on automated deployment, monitoring, failure, eventual

[1]https://martinfowler.com/bliki/AnemicDomainModel.html

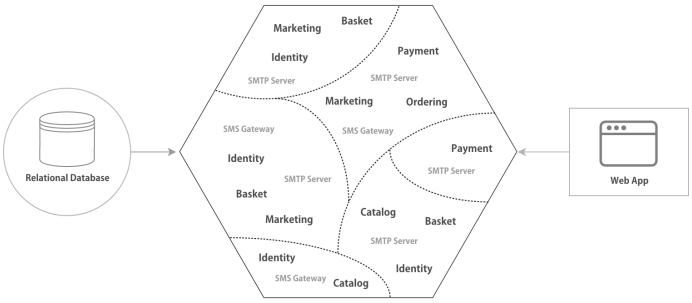

Fig. 4. A Traditional Monolithic Legacy Software System (Case Study)

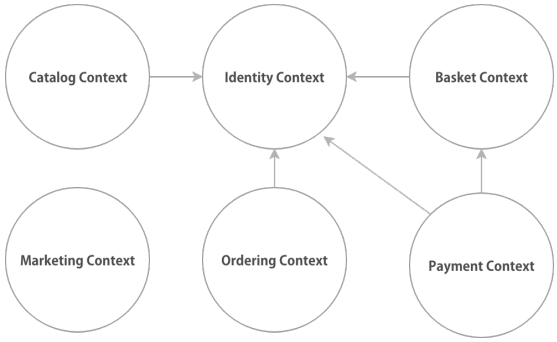

Fig. 5. A Context Map for the Monolithic Legacy Software System (Case Study)

consistency, and other factors introduced by a microservice architecture. For these reasons, we decided not to opt for the migration, and keep *ePromo* in its new modularized version.

At this point, we reached a preliminary list of lessons learned comprised of two main parts: *part 1* related to the restructuring of the legacy system to a modularized version and *part 2* related to migration of the modularized version to microservices. *Part 1* of the lessons learned are related to the (a) identification of candidate functionalities that can be modularized in legacy applications; (2) analysis of relationships and organizational dependencies in the legacy system; (3) identification of each domain and sub-domain. In the sequence, *part 2* of the lessons learned is related to the (4) selection of the candidates according to their importance to the domain and the application itself; (5) conversion of the candidate functionalities to microservices.

IV. The Case Study

The goal of this case study is to analyze an effective way to find candidate functionalities to be modularized in legacy applications to be later converted into microservices. *Target System for the Case Study*. Figure 4 illustrates a typical scenario of the *eShop* system. It is an online store in which users can browse a product catalog. The system provides functionalites such as user authentication, catalogue of products, special offers, and payments. All functionalities are implemented in the `PHP` programming language in a ”big module”, connected to a single relational database (MySQL). The system runs as a single artifact on a Nginx web server. The size of the source code increased dramatically over the years, as stakeholders asks for ever more changes and new functionalities. To deal with such requests, developers struggled to deliver new releases, which demanded ever more effort.

Part I - Migrating the Legacy System to a Modularized Version. We manually identified the candidate functionalities by navigating among the directories and files to find out the purpose of each artifact as was done for the pilot study. Figure 4 illustrates the entities `Identity`, `Basket`, `Marketing`, `Catalog`, `Ordering` and `Payment` related to the identified functionalities. This is the result of the first step aimed at identifying main functionalities and responsabilities in view of a tentative establishment of boundaries between them. Next, we planned to break down the main module into units. The key to this task was the use of bounded contexts ans their respective relationships, as represented in Figure 5. We applied in each bounded context the following DDD key concepts: *aggregate root*, *value objects* and *domain services*. These concepts help to manage domain complexity and ensures clarity of behavior within the domain model. After identifying contexts, we sorted them by level of complexity, starting with the simpler ones to validate the context mapping. We also placed the contexts into well-defined layers, expressing the domain model and business logic, eliminating dependencies on infrastructure, user interfaces and application logic, which often get mixed with it. We should concentrate all the code related to domain model in one layer, isolating it from the user interface, application and infrastructure parts [15]. In some cases, we can apply the *Strangler* pattern [6] to deal with the complexity of the module to be refactored.

A folder should be created for each of the bounded contexts and within each folder, three new folders should be added, one for each layer: `Domain`, `Application`, `Infrastructure`. They contain the source code necessary for this bounded context to work. It is crucial to consider the domain models and their invariants and to recognize entities, value objects and also aggregate roots. We should maintain the source code in these folders as described in the sequence. Folder `Application` contains all application services, command and command handlers. Folder `Domain` contains the classes with existing tatical patterns in the DDD, such as: `Entities`, `Value Objects`, `Domain Events`, `Repositories`, `Factories`. Folder `Infrastructure` provides technical capabilities to other parts of the application, isolating all domain logic from the details of the infrastructure layer. The latter contains, in more detail, the code for sending emails, post messages, store information in the database, process `HTTP` requests, make requests to other servers. Any structure and library related to ”the outside world”, such as network and file systems, should be used or called by the infrastructure layer.

Part II - Migrating from the Modularized version to Microservices At this point, our focus is the analysis of the previously developed context map and the assessment of

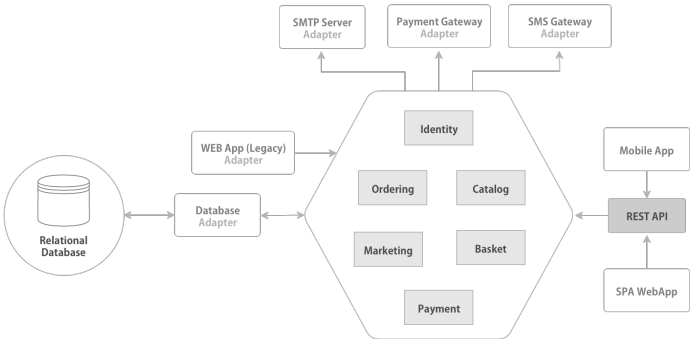

Fig. 6. An Evolved Monolithic Legacy System (Case Study)

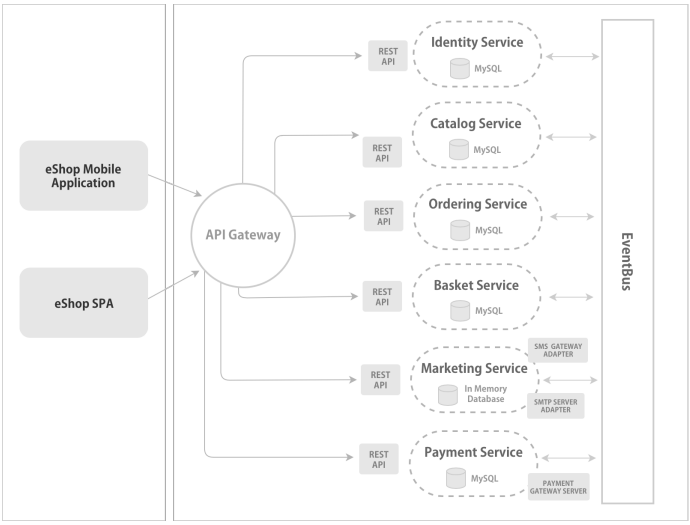

Fig. 7. A New Based Microservices Software System (Case Study)

the feasibility of decomposing each identified context into microservice candidates. In this case, during the analysis of the context map, it is required understanding and identifying the organizational relationships and dependencies. This is analogous to domain modeling, which can start relatively superficially and gradually increase levels of detail.

The most commonly used way to decompose an application into smaller parts is based on layered segmentation based on user interface, business logic and database responsibilities. However, this is prone to give rise to coupling between modules, causing the replication of business logic in the application layers [1] - *coupling* defines the degree of dependency between components or modules of an application. The microservice proposal to circumvent this problem entails segmenting the system into smaller parts with fewer responsibilities. In addition, it also considers domain, focus and application contexts, yielding a set of autonomous services, with reduced coupling.

In order to answer *SRQ2*, the bounded contexts from DDD are used to organize and identify microservices [17]. Many proponents of the microservice architecture use Eric Evans's DDD approach, as it offers a set of concepts and techniques that support the modularization in software systems. Among these tools, *Bounded Context* is used to identify and organize the microservices. Evans made the case for bounded contexts as facilitating the creation of smaller and more coherent components (models), which should not be shared across contexts. In the context map shown in Figure 5, the arrow is used to facilitate identification of upstream/downstream relationships between contexts. When a limited context has influence over another (due to factors of a less technical nature), provision of some service or information this relationship is considered upstream. However, the limited contexts that consume it comprise a downstream relationship [15].

Correct identification of bounded contexts using DDD and breaking a large system across them is an effective way of defining microservice boundaries. Newman points out that bounded contexts represent autonomous business domains (i.e., distinct business capabilities) and therefore are the appropriate starting point for identifying boundaries for microservices. Using DDD and bounded contexts lowers the chances of two microservices needing to share a model and corresponding data space, risking a tight coupling. Avoiding data sharing facilitates treating each microservice as an independent deployment unit. Independent deployment increases speed while still maintaining security within the overall system. DDD and bounded contexts seems to make a good process for designing components [14]. Note however, that it is still possible to use DDD and still end up with quite large components, which go against the principles of the microservice architecture. In sum, *smaller is better*.

An important service feature is its low number of responsibilities, which is reinforced by the definition of the *Single Responsibility Principle* (SRP) [18]. Each service must have a well-defined boundary between the modules, which should be independently created and published, through an automated deployment process. A team can work on one or several *Bounded Context*'s, with each serving as a foundation for one or several microservices. Changes and new features are supposed to related to just one *Bounded Context* and thus just one team [10].

Keeping all data on a single basis is contrary to the decentralized data management feature of microservices. The strategy is to move resources vertically by decoupling the primary feature along its data and redirect all front-end applications to the new APIs. Having multiple applications using the data from a centralized database is the primary lock to decouple the data along with the service.

Migrating data from an existing application is a complex process. It requires special care, which depends on the specific situation. During the migration of the *eShop* database, we decided to perform it in small chunks. We selected the tables related to each service and create a new database schema (MySQL) for the respective service. We then migrated them one by one. The database was not particularly large and this approach was applied without side effects. However, this approach may not be the most efficient, depending on the size of the database to be migrated. Each specific scenario must be analyzed in its terms. To perform the migration, we used

Doctrine Migrations[2]. Figure 7 shows the architecture of the new system based on microservices.

V. Lessons Learned

As a result of the experience gained in the two-phase study reported before, we identified four key challenges in the migration process. The first, is related to the identification of functionalities. This is not trivial, especially in cases of large modules through which functionalities are scattered and tangled among themselves. This is in fact a recurring issue already discussed in the literature [19]. The second challenge is the definition of optimal boundaries among candidate features for microservices. Once these limits are established, there follows the third challenge, to decide which will be converted to microservices. After this decision, we should face the fourth challenge, related to carefully analyze these candidate microservices regarding their respective granularity and respective cohesion.

The literature already addressed the *decomposition problem* for identifying modules, packages, components, and "traditional" services, mainly by means of clustering techniques upon design artifacts or source code. However, boundaries between modules defined using these approaches were too flexible and allowed software to evolve into instances of *Big Ball of Mud* [13].

Although much was written on the value of cohesive services and the power of bounded contexts, there appears to be a void in the guidance on how to identify these in practice [20]. The main issue is that those people trying to determine service boundaries are technologists looking for a technological solution but defining cohesive, capability-aligned service boundaries instead requires domain expertise. To accomplish this, a modelling exercise should be carried out independently of the specific technology used.

Applying the aforementioned strategies yielded multiple autonomous microservices, each with its own database. For communication between the microservices, we use `HTTP` communication mechanisms as API *Restful* and also asynchronous communication with an `EventBus` implementation, running *RabbitMQ*[3]. As shown in figure 7, each of the microservices now work with an independent relational database, except the `Marketing` service because it is an auxiliary service. For this one, we chose to use an in-memory database.

VI. Conclusions

Migrating a legacy application is often a hard and complex work. It rarely can be performed without significant effort. To the best of our knowledge, there are frameworks that can be used to support practitioners during the development (forward engineering) of microservice-based systems, such as *Spring Cloud* [4] and Hystrix [5], just to name a few. However, none of them provides full support to the three migration phases. To fill this gap, this paper present the lessons learned to support the stated migration. We believe that the availability these lessons learned can support and encourage practitioners from the industry and academia to perform this type of migration. Considering that these lessons learned is based on our experience on a specific two-phase study. We also plan to conduct a survey with practitioners from the industry to characterize their perception regarding challenges faced during this type of migration, characteristics of possible processes they may have used for this purpose and opinion about the lessons learned reported.

[2]https://www.doctrine-project.org/projects/migrations.html

[3]https://www.rabbitmq.com

[4]http://projects.spring.io/spring-cloud/

[5]https://github.com/Netflix/Hystrix

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[2] A. Singleton, "The economics of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 16–20, 2016.

[3] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, and T. M. Bohnert, "Self-managing cloud-native applications: Design, implementation, and experience," *Future Generation Computer Systems*, vol. 72, pp. 165–179, 2017.

[4] M. Kalske, N. Mkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *Current Trends in Web Engineering*. Springer, Cham, 2017, pp. 32–47.

[5] F. Leymann, U. Breitenbcher, S. Wagner, and J. Wettinger, "Native cloud applications: Why monolithic virtualization is not their foundation," in *Cloud Computing and Services Science*. Springer, Cham, 2016, pp. 16–40.

[6] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[7] C. Richardson, "Microservices: Decomposing applications for deployability and scalability," 2014.

[8] ——, "Pattern: Monolithic architecture," *Posjećeno*, vol. 15, p. 2016, 2014.

[9] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

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

[11] S. Millett, *Patterns, Principles and Practices of Domain-Driven Design*. John Wiley & Sons, 2015.

[12] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[13] J. O. Coplien and D. C. Schmidt, *Pattern languages of program design*. ACM Press/Addison-Wesley Publishing Co., 1995.

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

[15] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[16] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[17] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. " O'Reilly Media, Inc.", 2016.

[18] R. C. Martin, "The single responsibility principle," *The principles, patterns, and practices of Agile Software Development*, vol. 149, p. 154, 2002.

[19] H. Ossher and P. Tarr, "Multi-dimensional separation of concerns and the hyperspace approach," in *Software Architectures and Component Technology*. Springer, 2002, pp. 293–323.

[20] M. McLarty, "Designing a microservice system." [Online]. Available: http://www.apiacademy.co/designing-a-system-of-microservices