



**STATE UNIVERSITY OF CEARA  
SCIENCE AND TECHNOLOGY CENTER  
COMPUTER SCIENCE POSTGRADUATE PROGRAM  
ACADEMIC MASTER IN COMPUTER SCIENCE**

**MARX HARON GOMES BARBOSA**

**A PROCESS TO MIGRATE LEGACY SYSTEMS WITH BUSINESS RULES  
CONTAINED IN STORED PROCEDURES TO A MICROSERVICE-ORIENTED  
ARCHITECTURE**

**FORTALEZA – CEARÁ**

**2020**

MARX HARON GOMES BARBOSA

A PROCESS TO MIGRATE LEGACY SYSTEMS WITH BUSINESS RULES CONTAINED  
IN STORED PROCEDURES TO A MICROSERVICE-ORIENTED ARCHITECTURE

Dissertation presented to the Academic Master in Computer Science Course of the Computer Science Postgraduate Program of the Science and Technology Center of the State University of Ceara, as a partial requirement to obtain the title of Master in Computer Science. Concentration Area: Computer Science

Supervisor: Prof. Paulo Henrique Mendes Maia, PhD

FORTALEZA – CEARÁ

2020

Dados Internacionais de Catalogação na Publicação  
Universidade Estadual do Ceará  
Sistema de Bibliotecas

Barbosa, Marx Haron Gomes.

A process to migrate legacy systems with business rules contained in stored procedures to a microservice-oriented architecture [recurso eletrônico] / Marx Haron Gomes Barbosa. - 2020.  
57 f. : il.

Dissertação (Mestrado acadêmico) -  
Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Curso de Mestrado Acadêmico em Ciência da Computação, Fortaleza, 2020.

Orientação: Prof. Dr. Paulo Henrique Mendes Maia.

1. microservices. 2. legacy systems. 3. database. 4. extraction. 5. discovery. 6. process. I. Título.

MARX HARON GOMES BARBOSA

A PROCESS TO MIGRATE LEGACY SYSTEMS WITH BUSINESS RULES CONTAINED  
IN STORED PROCEDURES TO A MICROSERVICE-ORIENTED ARCHITECTURE

Dissertation presented to the Academic Master  
in Computer Science Course of the Computer  
Science Postgraduate Program of the Science  
and Technology Center of the State University  
of Ceara, as a partial requirement to obtain  
the title of Master in Computer Science.  
Concentration Area: Computer Science

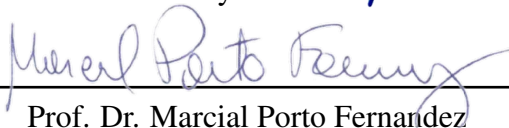
Approved on: 30/10/2020

EXAMINATION BOARD



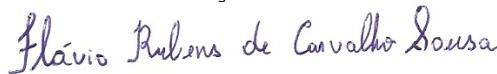
---

Prof. Paulo Henrique Mendes Maia, PhD (Supervisor)  
State University of Ceará – UECE



---

Prof. Dr. Marcial Porto Fernandez  
State University of Ceará – UECE



---

Prof. Dr. Flávio Rubens de Carvalho Sousa  
Federal University of Ceará – UFC

To my lovely family that is the reason of my all  
effort and in memoriam of my father.

## **ACKNOWLEDGEMENTS**

First of all I thanks God for giving me health and perseverance tracing in this journey.

Also I am very pleased to know that in this entire path I had the support of my lovely family specially from my wife that even knowing nothing about IT stuffs, she has always been following closely and interested about my research project. Thank you my love, so much!

To my mother that prays for me everyday, and to my lovely father (in memorium). I miss you!

I can not to forget to express my huge admiration to all the professors that I had the opportunity to work through this time that I had in MACC, specially my supervisor and friend Paulo Mendes.

It was an honor to work and learn with you all.

For the last but not least I'd like to say thank to all my friends that supported me and are very happy with my achievement.

“Veni, vidi, vici.” (I came, I saw, I conquered)  
(Julius Caesar)

## ABSTRACT

During the 1980 and 1990's decades, relational database management systems arose as an alternative to implement and store application business logic due to its robustness. Many of those legacy systems suffer from several problems such as low scalability, database vendor lock-in, and complex maintenance and evolution. With the success of lightweight virtualization techniques and new distributed architectures, mainly the microservices, companies are migrating legacy systems to this architectural style. Although several studies have proposed migration processes and reported migration experience to microservices, to the best of our knowledge, none of them has addressed systems whose business rules are implemented in database artifacts, particularly stored procedures. Therefore, this paper presents a process to identify microservice candidates from application business rules implemented in stored procedures. We applied the process to a real large scale system, for which 362 business rules were mapped and 13 microservices were identified. In addition, the process helped to find out many duplicated pieces of code, thus also improving the system maintainability.

**Keywords:** microservices. legacy systems. database. extraction. discovery. process



## SUMMARY

<b>1</b>	<b>INTRODUCTION</b>	<b>10</b>
<b>1.1</b>	<b>Objectives</b>	<b>12</b>
<b>1.1.1</b>	<b>Overview of the dissertation</b>	<b>12</b>
<b>2</b>	<b>BACKGROUND</b>	<b>13</b>
<b>2.1</b>	<b>Legacy monolithic systems</b>	<b>13</b>
<b>2.2</b>	<b>Microservices</b>	<b>15</b>
<b>2.2.1</b>	<b>Main characteristics</b>	<b>16</b>
<b>2.2.2</b>	<b>Microservice mechanisms</b>	<b>18</b>
<b>2.2.3</b>	<b>Strangler pattern</b>	<b>20</b>
<b>2.3</b>	<b>Relational database systems and artifacts</b>	<b>21</b>
<b>2.3.1</b>	<b>Stored procedures</b>	<b>22</b>
<b>2.3.2</b>	<b>Triggers</b>	<b>23</b>
<b>2.3.3</b>	<b>Functions</b>	<b>24</b>
<b>3</b>	<b>RELATED WORK</b>	<b>25</b>
<b>4</b>	<b>THE PROPOSED PROCESS</b>	<b>28</b>
<b>4.1</b>	<b>Microservice discovery and extraction</b>	<b>29</b>
<b>4.1.1</b>	<b>System requirement selection and stored procedure mapping</b>	<b>29</b>
<b>4.1.2</b>	<b>Microservice candidates discovery</b>	<b>31</b>
<b>4.1.3</b>	<b>Microservice candidates validation</b>	<b>32</b>
<b>4.2</b>	<b>Architecture definition and microservice creation</b>	<b>32</b>
<b>4.3</b>	<b>System refactoring</b>	<b>34</b>
<b>5</b>	<b>PROOF OF CONCEPT</b>	<b>36</b>
<b>5.1</b>	<b>Legal, ethical and intellectual property issues</b>	<b>36</b>
<b>5.2</b>	<b>Applying the proposed process</b>	<b>37</b>
<b>5.2.1</b>	<b>Phase 1: microservice discovery and extraction</b>	<b>37</b>
<b>5.2.2</b>	<b>Phase 2: architecture definition and microservice creation</b>	<b>43</b>
<b>5.2.2.1</b>	<b>Implementing microservices candidates</b>	<b>43</b>
<b>5.2.2.2</b>	<b>Setting up microservice infrastructure</b>	<b>46</b>
<b>5.2.3</b>	<b>Phase 3: system refactoring</b>	<b>47</b>
<b>6</b>	<b>DISCUSSION AND LESSONS LEARNED</b>	<b>50</b>
<b>7</b>	<b>CONCLUSION</b>	<b>52</b>

<b>7.1</b>	<b>Limitation . . . . .</b>	<b>52</b>
<b>7.2</b>	<b>Publication . . . . .</b>	<b>53</b>
<b>7.3</b>	<b>Future work . . . . .</b>	<b>54</b>
	<b>REFERENCES . . . . .</b>	<b>55</b>

## 1 INTRODUCTION

During the 1980 and 1990's decades, with the increased popularity of object-oriented languages, techniques and frameworks, and the consolidation of relational database management systems (RDBMS), there was a boost in the development of client-server information systems (FONG; HUI, 1999), which were implemented using IDEs like Visual Basic, Delphi, and SQLWindows. Products like IBM DB2 and Oracle Database were released and became dominant in data centers of mid-to-large corporations.

At that time, many of the existing information systems were structured using a monolithic architecture based on the Model-View-Controller (MVC) pattern (BUSCHMANN; HENNEY, 1996) and implemented following an object-oriented development approach, such as the Rational Unified Process (RUP) (RATIONAL, 2011). Nonetheless, those systems struggled with performance issues when processing large amounts of data.

As database servers became more robust and worked as stand-alone services in dedicated machines, to solve that problem, system analysts migrated some part of the application business logic to database artifacts, such as stored procedures and triggers. This way, the application code was responsible for dealing mainly with requests from user interfaces and for redirecting them to the correct database artifact, releasing processing on the application server-side and transferring those responsibilities to the database server.

Using stored procedures (SP) was the most appropriate approach in several cases because they encapsulate parts of the application logic, improving performance by both reducing the network traffic between client applications and servers and optimizing the SQL statement execution (LAHIO F. LAUX; DERVOS, 2017). Therefore, the performance problem was transferred to the database level, making the application code simpler and more efficient (ELMASRI; NAVATHE, 2015).

Although, on the one hand, implementing business rules in SPs solved the performance issue at that time, on the other hand it generated problems such as scalability, code duplication, difficulty in handling exceptions, and vendor lock-in. Consequently, some activities, like system maintenance, debugging and evolution, became very challenging and tiring. With the adoption of NoSQL databases from the 2000s, companies realized that using stored procedures for new projects was a bad and risky design decision.

One approach to tackle the scalability problem was creating clusters of servers to increase the processing capacity and the availability of the service, but that approach had a

physical hardware limitation and other problems, like maintenance and portability, were still happening.

To migrate database artifacts deployed from a legacy system to another specific database vendor, it was necessary to completely rewrite the artifact code to adjust it to the other vendor-directed syntax, since each DB worked with different programming languages for its artifacts, which configures a vendor lock-in. This has implications for maintenance, because everytime that a single rule changes, it must be rewritten in every different syntax supported by the system.

With the success of virtualization techniques, like containers, and new distributed architectures, companies are migrating legacy systems to those new hardware and software technologies in order to mitigate the aforementioned problems. Particularly, microservice-based architectures have been gaining attention as a better solution for efficiently scaling computational resources (NADAREISHVILI *et al.*, 2016). Microservices are a recent architectural style to develop a single application as a collection of independent, well-defined, and intercommunicating services that communicate with each other through lightweight mechanisms (LEWIS; FOWLER, 2014)(PAHL; JAMSHIDI, 2016). Usually each microservice has its own database that concerns only a bounded context of the system, thus helping to tackle software complexity and scalability(KNOCHE; HASSELBRING, 2018)(Bucchiarone *et al.*, 2018).

Many studies have proposed processes and frameworks to migrate legacy systems to a microservice-based architecture, aiming at creating low coupling and high reusable systems (KNOCHE; HASSELBRING, 2018)(Bucchiarone *et al.*, 2018)(BALALAIE; HEYDARNOORI; JAMSHIDI, 2016)(LEVCOVITZ; TERRA; VALENTE, 2016). However, to the best of our knowledge, none of them has addressed systems whose most of the business logic are contained in database artifacts. Evolving such systems is challenging since the business logic is spread throughout several different artifacts, thus turning the identification process a non-trivial task.

Therefore, in this work it is proposed a novel process to identify and extract microservices candidates from business logic implemented in database artifacts, more specifically stored procedures. It is a three-phase process that consists of (i) an initial process to discover, isolate and evaluate code slices to be potential microservice candidates, that has three main phases: (a) *system requirement selection and stored procedure mapping*, in which some requirements and their corresponding system features are analysed to discover all SPs related to those selected system requirements, (b) *microservice candidates discovery*, in which microservice candidates are identified from a thorough analysis of the stored procedures, and (c) *microservice candidates*

*validation*, which aims at removing unnecessary microservices and refactoring the remaining ones, when appropriate. The second phase consists of (ii) proposing architectural support and implementing the emerged microservices, and (iii) suggesting a code refactoring to evolve the system architecture.

The whole process was evaluated by means of a proof of concept applied to a real large scale system, for which 362 business rules were mapped and 13 microservices were identified. The steps forward were to implement the identified microservices using as inputs the results of the previous process phases, applying microservices support architecture patterns, and guiding the code refactoring to evolve the legacy system to use them. As a result, the process helped to find out many duplicated pieces of code, thus also improving the system maintainability.

## **1.1 Objectives**

The main objective of this work is to detail a process to discover and extract microservices from business logic implemented in a legacy system database. To accomplish that, the following objectives must be achieved:

- a) To model the phases of the process, including their inputs and outputs;
- b) To describe the steps of each phase to accomplish the process;
- c) To evaluate the application of the process in a proof of concept.

### **1.1.1 Overview of the dissertation**

The remainder of this work is as follows. Chapter 2 deals with the literature review used to support research. Chapter 3 presents and compares the main related work. Following, Chapter 4 details the whole process proposed by this work. Chapter 5 presents and discusses the proof of concept carried out to evaluate the proposed process. Finally, Chapters 6 and 7 draw the lessons learned, conclusions and future work.

## 2 BACKGROUND

This chapter presents the literature review carried out to define the main concepts upon which this dissertation is built. Section 2.1 discusses the main concepts of legacy systems and their relationship to database-implemented artifacts. Section 2.2 presents the microservice concepts and details the main microservice supporting tools and mechanisms. Section 2.3 presents relational database systems and artifacts concepts for legacy systems and microservice-oriented architecture.

### 2.1 Legacy monolithic systems

A legacy system represents a system built on a highly complex coupled monolithic architecture. These systems were widely used in the 1980s and 1990s, built primarily using the mainframe architecture. Over the years, those systems have evolved, but the concept of monolithic structure has remained.

In the beginning of the 1990s, the client-server model imperated as the best architectural style when considering system engineering development techniques. Almost all enterprise systems were developed using a structured programming language, in an environment that consumed high level of resources (LINGER; MILLS; WITT, 1979) (MILLS, 1986). The server-side system modules were robust and became complex and difficult to maintain, with compiled source code consuming large portions of memory and processing (OROUMCHIAN, 2003). As the data processing increased, those computational resources turned to be a bottleneck in the system performance.

After about 10 years, in the beginning of the 2000s, with the growing popularity of object-oriented languages, specially Java, techniques and frameworks, and the consolidation of relational database management systems (RDMS), there was an increase in the development of client-server information systems (FONG; HUI, 1999). Many of those systems were structured and implemented using a monolithic architecture based on the Model-View-Controller (MVC) pattern (BUSCHMANN; HENNEY, 1996) and followed an object-oriented development approach, such as the Rational Unified Process (RUP) (RATIONAL, 2011).

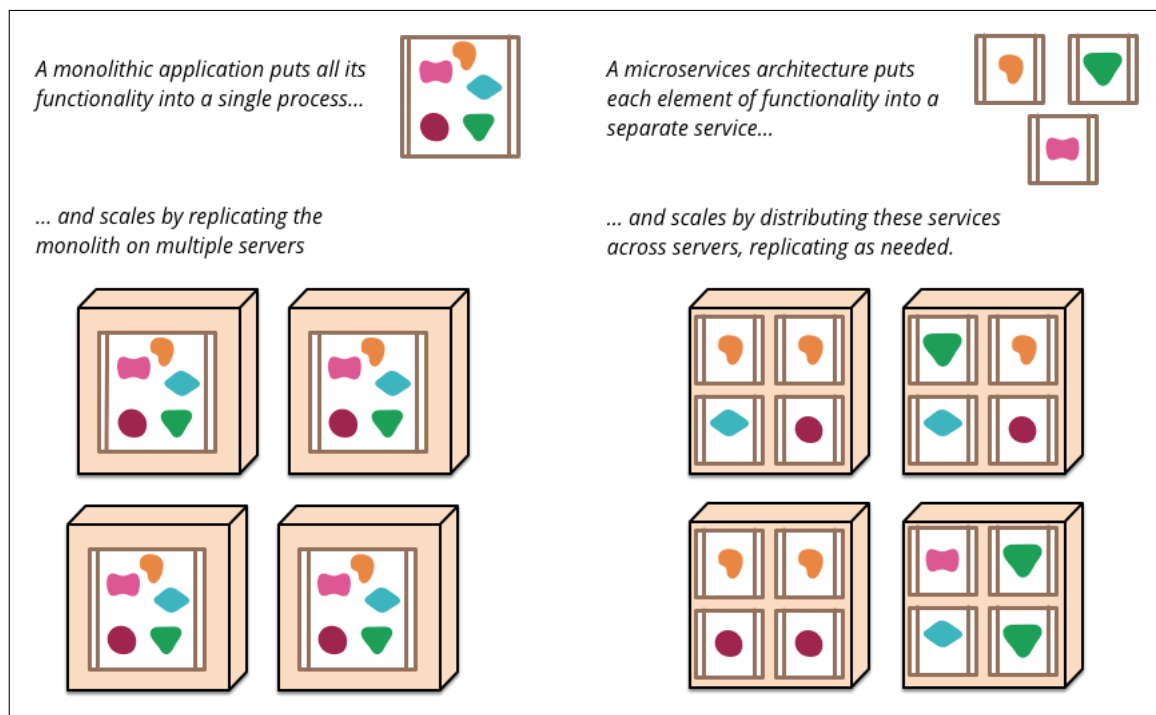
Then, in the decade of 2010, with the arrival of SOA, many systems changed to that new architectural style. However, they have in common the fact that, despite the increase in modularity, they continue to have a single deployment artifact, which constitutes a monolithic architecture.

That type of approach brought many barriers that software engineers had to overcome in order to make these systems meeting the increasing demand for availability, scalability, fault tolerance, modularization, decoupling and maintainability, which in robust, complex and tightly coupled systems means a major investment in their evolution (SARKAR *et al.*, 2009).

However, the evolution and improvement of such legacy systems is a complex task since they had take a lot of time and effort to develop, evolve and maintain, as well as handle large databases of sensitive information that directly address customers needs. According to Levcovitz, Terra and Valente (2016) and Markus and Tanis (2000) “a major challenge nowadays on enterprise software development is to evolve monolithic system on tight business schedules, target budget, but keeping quality, availability, and reliability”.

With the recent advance of cloud computing, the microservice technology arises as an option for evolution and incremental rewriting of those legacy systems. Specially when addressing the evolution of those systems, microservices add a high availability and easy-to-maintain componentization (see Figure 1) that reduces the risk of system-wide errors and eases traceability and testing (KNOCHE; HASSELBRING, 2018).

Figure 1 – Monoliths and Microservices.



Source:(LEWIS; FOWLER, 2014)

In Figure 1, Lewis and Fowler (2014) make an abstraction about the representation of a monolithic system, where all its functionalities are implemented in a cohesive way, highly

coupled, and being distributed as a single block. If there is a need to scale the monolithic system, it would follow a model of total replication on other servers such that even if only a part of the system had this need, the entire system should be replicated. In contrast, in a microservice-oriented architecture, in which each service represents an isolated functionality, being a decoupled and highly scalable model, where replication would be limited to a specific functionality and not the entire system, as in the monolithic case.

## 2.2 Microservices

The concept of microservice is tied to a change in the system building architecture that supports the independence of modules that aggregate unique, high scalability and autonomy features (LEWIS; FOWLER, 2014). Its definition came in early 2011, when the industry sought options for SOA as the primary data communication platform between system units aiming the loose coupling of units and the decrease in complexity involved in this implementation.

One might think that microservice would be an extension of the concepts of SOA because it was its origin. However, there exists basic conceptual differences between them. The first, and perhaps most important difference, is that microservices are implemented to be independent units, which either alone or orchestrated, meet the business requirements of a specific system (NEWMAN, 2015). On the other hand, in SOA, in most cases, there is a high coupling of different services to meet this requirement. According to (Bucchiarone *et al.*, 2018), “SOA has no focus on independent deployment units and related consequences, it is simply an approach for business-to-business intercommunication”.

Supported by the recent advances of cloud computing, the microservice technology arises as an option for evolution and incremental modernization of legacy systems by using a high availability and easy-to-maintain componentization that reduces the risk of system errors and eases traceability and testing (KNOCHE; HASSELBRING, 2018). The architectural change proposed by microservices also popularized a new communication concept that replaced the SOAP model used by webservices, predominant in SOA architecture, for lightweight REST communication (TAIBI; LENARDUZZI; PAHL, 2018). The emergence and consolidation of specific technologies has contributed to the implementation and evolution of microservices such as containers, discovery services, orchestration and monitoring, fault tolerance reporting, continuous deployment, DevOps (BASS; WEBER; ZHU, 2015), chaos engineering, and so on (JAMSHIDI *et al.*, 2018)(TAIBI; LENARDUZZI; PAHL, 2018).



When it comes to microservice, it is not possible to dissociate from an architecture that supports it. One can then think of an architectural pattern suitable for using microservices, but that pattern can vary greatly depending on each context. Some characteristics in the chosen architecture must be observed so that it can be characterized as a micro-service oriented architecture (LEWIS; FOWLER, 2014).

### **2.2.1 Main characteristics**

The first characteristic sought in an architecture oriented to microservices is the concept of componentization through services. In a traditional architecture, the concept of components strongly indicates a reuse of self-contained modules, such as shared libraries. The same concept can be extended to services. Since services can be distributed independently, they are more efficient than libraries because a change in a library would require the publication of the entire system, but when it comes to changing a service, just publish it. This can lead to an increased need for coordination between services that must be mitigated with efficient mechanisms in service contracts and publishing interfaces.

The second characteristic to be observed in a microservice oriented architecture is its organization around business capabilities. This indicates a model of development teams now focused on a certain part of the business, as opposed to a single team stratified in layers of a single system treating the business as a single block, where the interaction between its components follows Conway's Law (CONWAY, 1968) that dictates that the structural organization of the system copies the structure of the organization itself. In a microservice-oriented architecture, these teams are multifunctional and each team deals with a specific part of the business, modularizing the system into different parts, thus facilitating its maintenance.

A third characteristic sought is the change in the product delivery paradigm and not a project. In traditional architectures, the system is seen as a project, with a defined lifecycle, while in microservice-oriented architectures the service is a product and the development team is responsible for its implementation, delivery and maintenance throughout its entire lifecycle. Here, it is possible to see the concept of granularity of the product in relation to the part of the business that it must serve and not to the whole that an entire system would propose.

When considering about integration between different processes in a traditional architecture, one can think about an intelligent communication bus, in which all the mechanisms of message exchange, fault tolerance, routing, choreography and application of business rule are

present. Applications based on a microservice architecture try to avoid this approach. Decoupling and independence between microservices is essential for issues of maintenance and evolution of services. Choreography between services is essential and communication between them must be through a lightweight mechanism as opposed to complex protocols, opting for communication protocols based on the exchange of web messages (HTTP request-response).

Another characteristic seen in microservices-oriented architectures is decentralized governance. Monolithic architectures, for example, basically use a single technological platform, which can become a very restrictive choice, as it limits the possibilities of evolution and maintenance to the limits of the chosen technology. Microservices can be implemented in different technologies, each one being specific to the characteristic that the service requires, without the need for standardization or overloading the service with unnecessary characteristics. The concept of reuse and sharing is also present in this concept of decentralized governance, in which open source practices between development communities are encouraged. Another observed point of the decentralization of governance, in the use of an architecture oriented to microservices, is the transfer of responsibilities of the layers of management of the companies on the products they offer to the development teams, who become directly responsible for all aspects software including operation on 24/7 models.

The decentralization of data management is also a characteristic to be observed. Typically, in monolithic systems, this is done centrally in a single data domain that encompasses all business needs. In general, a single centralized database is shared among the various functionalities of the system. Microservices that serve a specific niche in the business tend to maintain their own specific database to serve that niche. Or even, a mixture of approaches to reorganize this information using different storage technologies and different databases, an approach known as Polyglot Persistence. This concept of independence and self-containment of microservices, regarding data persistence and integrity, brings the need to guarantee the consistency of this information between multiple devices, where access and stability are not always guaranteed, which can cause a transactional problem. Then, there is a preference for a model that emphasizes non-transactional coordination between these services, requiring mechanisms to be implemented to manage this data inconsistency by the microservice development teams.

Infrastructure automation, specially when it comes to deployment, is a concept widely used both for monolithic systems and for microservices. The difference basically lies in the availability of the service and its maintenance. New versions of a functionality in a monolithic system imply a unique process of deploying the system in a comprehensive way,

as well as ensuring the availability of a certain functionality is to ensure that the entire system is alive. When considering microservices, that approach changes, as each service has its own distinct instance and deployment, and in the case of failure, only the impacted service is replaced as a contingency, thus avoiding system unavailability.

This concept of fault tolerance leads to a new characteristic of microservices oriented architectures, the pressing need to constantly monitor the execution of each service in a different way and the containment measures that must be carried out to guarantee the availability of the business.

No less important is to think about a design of the system in order to facilitate its evolution and maintenance. Perhaps it is the most unique feature of microservice-oriented architecture. Herein lies the need without thinking about how the system will be modularized, componentized, whether as a whole or in parts. “This emphasis on replaceability is a special case of a more general principle of modular design, which is to drive modularity through the pattern of change ” (LEWIS; FOWLER, 2014). Thus, the strategy is to group things that undergo simultaneous changes in the same module, and preferably in the same service if applicable.

### **2.2.2 Microservice mechanisms**

The microservice architecture has boosted a system to be as scalable as possible, in addition to rapid evolution, maintenance and responsiveness (PAHL; JAMSHIDI; ZIMMERMANN, 2017). However, that new paradigm brought the need of a new set of supporting technologies that enabled its application, such as service discovery, service registration, API gateway, service orchestrator, monitoring tools and service balancing that will be mentioned below.

The service independence concept (NEWMAN, 2015) is a fundamental part of the architecture structure, which was designed in a way that the service could be activated, replaced and stopped without interruption of other services. That was possible due to the concept of containers (e.g. Docker, LXC and rkt) (JAMSHIDI *et al.*, 2018). Containers can encapsulate images (versions) of a service and provide a basic infrastructure for accessing it. It is possible to monitor each active container and to perform interventions in case of faulty behavior, to update services at run time, as well as to synchronize their execution. Examples of service monitoring tools are Graphite <sup>1</sup> and Prometheus <sup>2</sup>.

---

<sup>1</sup> <https://graphiteapp.org/>

<sup>2</sup> <https://prometheus.io/>

The synchronization, necessary for the service execution, is in charge of an orchestrator. The orchestrator is another tool used in microservice-oriented architecture whose main role is to manage (coordinate) the execution between microservices to meet a specific business requirement (JAMSHIDI *et al.*, 2018). Examples of service orchestrators are Docker Swarm <sup>3</sup>, Marathon <sup>4</sup> and Kubernetes <sup>5</sup>.

API Gateway is the mechanism by which communication between the system and microservices is enabled. It serves as a router to which microservice a particular request should be redirected. According to Taibi et al (2018), “API Gateway is the entry point of the system that routes the requests to the appropriate microservices, also invoking multiple microservices and aggregating results”. It can also be used as a means of authentication, monitoring and in some cases as a load manager (TAIBI; LENARDUZZI; PAHL, 2018). Examples of API Gateway tools are Zuul <sup>6</sup>, Kong <sup>7</sup>, AWS <sup>8</sup>, and TyK <sup>9</sup>.

One advantage of using API Gateway in legacy system migration processes is the ease of redesigning the legacy system architecture to access the new microservice architecture. Also according to Taibi et al (2018), “...migrations from monolithic systems tend to be architected with an API-Gateway architecture, probably due to the fact that, since the systems need to be completely re-developed and rearchitected, this was done directly with this approach”. The use of an API Gateway is a good resource to apply the strangler pattern to migrate from monolithic to microservices architecture.

Since a microservice may be not available at the run time for some reason, or if there is a need to scale up the service to meet growing demand (PAHL; JAMSHIDI; ZIMMERMANN, 2017), the need of a service balancing mechanism arises (TAIBI; LENARDUZZI; PAHL, 2018), so that the service availability can be guaranteed. Examples of such tools are Eureka <sup>10</sup>, Kubernetes, Finagle <sup>11</sup> and Proxygen <sup>12</sup>.

Considering a wide portfolio of microservices, remote services, available in several servers that communicate in different ways, and once knowing which information is required, an efficient way is needed to identify which microservice is capable of providing this information.

<sup>3</sup> <https://docs.docker.com/engine/swarm/>

<sup>4</sup> <https://mesosphere.github.io/marathon/>

<sup>5</sup> <https://kubernetes.io>

<sup>6</sup> <https://github.com/Netflix/zuul>

<sup>7</sup> <https://konghq.com/kong/>

<sup>8</sup> <https://aws.amazon.com/pt/api-gateway/>

<sup>9</sup> <https://tyk.io/>

<sup>10</sup> <https://github.com/Netflix/eureka>

<sup>11</sup> <https://twitter.github.io/finagle/>

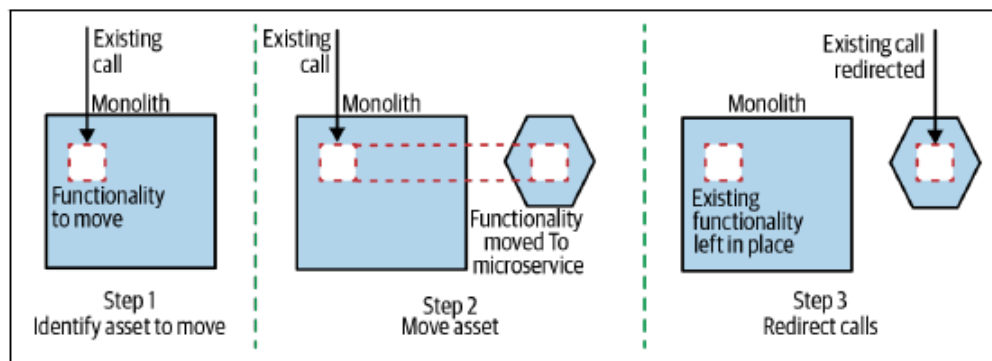
<sup>12</sup> <https://github.com/facebook/proxygen>

This is possible through a microservice discovery tool, as address Taibi, Lenarduzzi e Pahl (2018), “the service discovery dynamically supports the resolution of DNS address into IP addresses”. That tool registers information about the microservice and its properties that are available. Eureka, Consul <sup>13</sup> and Zookeeper <sup>14</sup> are examples of service discovery tools.

### 2.2.3 Strangler pattern

Martin Fowler first captured this pattern by observing the fig that grew on the top of trees and descended toward to the ground to take the root, gradually enveloping the support tree and leading it to die (FOWLER, 2004).

Figure 2 – An overview of Strangler Pattern.



Source:(NEWMAN, 2019)

This pattern is commonly used for migration from monolithic systems to microservices and consists of incrementally encapsulating parts of the system in one or more microservices. After those parts are encapsulated, when requested to the original system, it is redirected to the specific microservice that implements the same original behavior. Now the encapsulated parts can be isolated in the monolithic system and can no longer be used (see Figure 2). Other functionalities that have not yet been encapsulated continue to run using the monolithic system. With the continuous migration of parts of the monolithic system to microservices, the strangler pattern will refactor and disable the original monolithic system when the entire system has been converted (or the modules that matter) to the new architecture (NEWMAN, 2019).

<sup>13</sup> <https://www.consul.io/>

<sup>14</sup> <https://zookeeper.apache.org/>

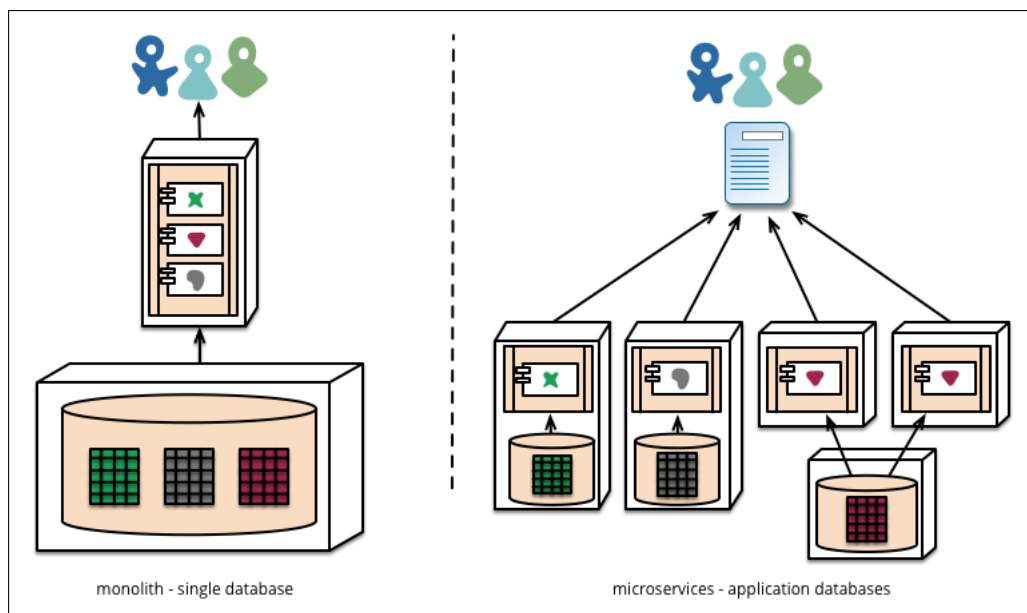
### 2.3 Relational database systems and artifacts

Data storage is always a concern when defining which architecture the system will have (TAIBI; LENARDUZZI; PAHL, 2018). Most legacy systems were designed with a centralized data model, in which there were shared servers that hosted database management systems and the databases themselves (ELMASRI; NAVATHE, 2015). The servers were robust machines that require a high infrastructure for support.

Centralized databases, in turn, shared information between system modules as well as with other systems. Although they are usually proprietary and difficult to port, scale and maintain (ELMASRI; NAVATHE, 2015), that architecture was widely used as those machines had high performance, which in the early 1980s and 1990s was a difficult requirement given the available software and hardware technologies. Thus, many software engineers have chosen to migrate client-side processing to server-side processing, including much of the business logic in database artifacts residing in those centralized databases (ELMASRI; NAVATHE, 2015).

Those artifacts consisted of interpreted codes that followed structured, procedural, and sequential logic (stored procedures, functions, triggers, indexes, metadata among others) that worked directly with the database entities (tables and views) providing fast access and information processing (ELMASRI; NAVATHE, 2015).

Figure 3 – Monoliths x Microservices Data Storage.



Source:(LEWIS; FOWLER, 2014)

Some legacy systems have been migrated to decentralized remote databases over

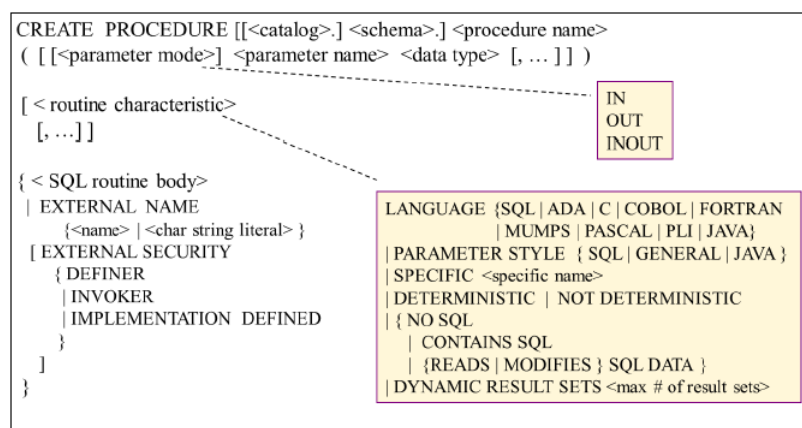
time, but now they have to deal with data synchronization, availability, and more problems. With the rise of technologies that supported a more robust software client, with acceptable performance requirements that no longer justify the need for a high-throughput data server, most of the processing could be done on the client itself, so it was necessary to migrate in the opposite direction, leaving only the storage task to the data servers.

With the advent of the microservice-oriented architecture (LEWIS; FOWLER, 2014), the strategy of data storage has changed again. Microservices may use some different architectural patterns to address data storage according to the literature, such as one database per service, i.e., each microservice would access its own database, and a shared database (see Figure 3), which is widely used for legacy system migration since the legacy database should not be modified for its use by microservices (TAIBI; LENARDUZZI; PAHL, 2018). Three of the main database artifacts are described in the next sections: stored procedures, triggers and functions.

### 2.3.1 Stored procedures

Stored procedures are database artifacts used to store and process parts of application logic. They are written using a database vendor specific language, but interpret the native SQL language. In addition, SPs optimize SQL code execution by storing the execution plan for future calls, which minimizes its processing.

Figure 4 – Excerpt of ISO SQL Stored Procedure syntax.



Source:(MELTON, 2002)

SPs can improve the performance of legacy applications by running part of database server-side business processes. They can be parameterized and use polymorphism techniques, facilitating its (re)use in the application code, since it is independent of the client application

language. They use structured procedural language, which facilitates their reading, as well as transactional control (LAHIO F. LAUX; DERVOS, 2017).

The structure of a stored procedure varies according to the database provider, but all follow an ISO SQL standard in their composition. Figure 4 represents a standard syntax for creating generic stored procedures (MELTON, 2002). The focus of this work is on analysing the parameters and the body of the SQL routines to identify microservice candidates.

### 2.3.2 Triggers

Trigger is another type of database artifact that can also control the behavior of the system in terms of business rules. It cannot be accessed directly by the application, but it monitors events in the table to which it is linked, and is programmatically executed depending on the expected event (INSERT, UPDATE or DELETE). Query events are not able to execute a trigger in the database. In its body there is a sequence of commands that can implement business rules in the same way as stored procedures.

The structure of a trigger varies according to the database provider, but all follow an ISO SQL standard in their composition. Figure 5 represents a standard syntax for creating generic triggers (MELTON, 2002). In the present work, a process for discovering and extracting rules in triggers was not addressed, but it can be extended in future work in this regard.

Figure 5 – Excerpt of ISO SQL Trigger syntax.

```
CREATE TRIGGER <trigger name>
{BEFORE | AFTER | INSTEAD OF } <trigger event> ON <table name>
[REFERENCING OLD AS <old alias> ]
[REFERENCING NEW AS <new alias> ]
<triggered action>

Where
<trigger event> ::= INSERT | DELETE | UPDATE [OF <column list>]
<triggered action> ::=
[FOR EACH {ROW | STATEMENT}]
[ WHEN ( <SEARCH CONDITION> ) ]
{<SQL statement> |
  BEGIN ATOMIC
    {<SQL statement>;}...
  END
}
```

----- Action Granularity  
 ----- Action Condition  
 ----- Action Body

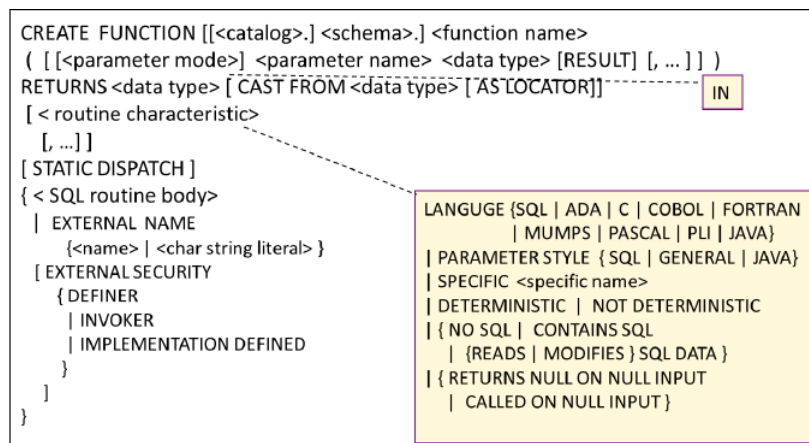
Source:(MELTON, 2002)



### 2.3.3 Functions

The SQL dialects of the various database vendors have mostly native functions that perform aggregation, arithmetic, temporal and string procedures defined in the SQL standard. They also provide a means of extending the language by user-defined functions (UDF), also called Stored Functions, which are often called from SQL statements. As referred by Melton, “...Compared with the stored procedures, a stored function returns only one SQL value (which in some implementations may also be a result set in form of table or opened cursor) the type of which is defined by the RETURNS clause” (MELTON, 2002).

Figure 6 – Excerpt of ISO SQL Function syntax.



Source:(MELTON, 2002)

The structure of a function varies according to the database provider, but all follow an ISO SQL standard in their composition. Figure 6 represents a standard syntax for creating generic functions (MELTON, 2002). As the same way as triggers, in the present work, the proposed process does not address functions, but it can be extended in future work in this regard.

### 3 RELATED WORK

In this chapter, some work that use (micro)services as the target independent modules for which legacy monolithic system's architecture have been migrated to are discussed and compared to the approached proposed in this work.

In (NAVEA *et al.*, 2019) a proposal is presented on how to organize and prioritize the architectural needs of the migration process from the legacy system to a microservice architecture, considering the characteristics of the legacy system, business needs, existing infrastructure, maintenance and future developments. ATAM (Architectural Trade-off Analysis Method) is used, method that allows to determine how well an architecture meets specific quality objectives pointed out by the stakeholders, but also to determine how the quality objectives and the architectural decisions are interrelated. At the end of the study, objectives were proposed to be achieved in line with a certain architectural model that should be used with the implementation of microservices, in order to migrate the legacy system incrementally aiming at the fulfillment of all the requirements raised during the application of ATAM. The proposal is interesting and could be used as a step prior to the application of the process proposed in this work, since one cannot deciding on migrating to microservices without considering the architectural structure that will support this migration.

The experience report described in (GOUIGOUX; TAMZALIT, 2019) corroborates the proposal presented in (NAVEA *et al.*, 2019) in which the migration from a monolithic legacy system must be a mainly functional process as opposed to a purely technical proposal. The functional proposal exemplified in the migration report from a monolithic system, focuses on business issues, defined through DDD (Domain Driven Design) approaches, migration strategies where the main point would be the integration between different business scenarios and how to benefit from an architecture based on microservices, and once these contexts are defined, the choice of the appropriate tools for the definition of the architecture to be used and how the microservices would be implemented. Once again, the need to observe the issue of business requirements and the existing infrastructure, in order to define a migration from a monolithic system, emerges as a point of intersection between several authors, which reinforces the need presented here in this work to understand these requirements previously to decide what will be migrated and how this migration will occur for an architecture based on microservices.

In (FRITZSCH *et al.*, 2019) a study carried out on 14 monolithic systems distributed in 10 companies, which at some point opted to use a microservice architecture to evolve these

systems, presented as significant results that most companies did not use a structured mechanism for migration (service discovery) from the system to the new architecture. It also reported difficulties in how to make the cut in the system of which functionalities would be migrated to microservices. The most successful approaches were those based on the use of the strangulation model gradually, in a strategy to keep both the legacy system and the “new” system in parallel. It was also reported that the functional model presented in (NAVEA *et al.*, 2019) and (GOUIGOUX; TAMZALIT, 2019) was considered to be a good strategy for modularizing and grouping services.

In (NUNES; SANTOS; SILVA, 2019), the proposal is to carry out a semi-automated analysis of the monolithic system source code, to group domain entities in transactional contexts performed by them, thus proposing the creation of microservices that would implement this grouping structure. A limitation that arises in this approach is the need for monolithic systems to follow an MVC (Model-View-Controller) structure, in which the main entry in the first phase of the analysis would be the control classes. The grouping mechanism checks the sharing of domain entities between controllers and assigns weights to those relationships, but it does not suggest weights for business requirements. Another difference in the proposal is that it does not consider business requirements implemented in other classes, much less in database artifacts. It also makes no mention of the microservice architecture that will be used to adapt to this refactoring of the monolithic system, it only indicates that each microservice will use its own database.

Gysel *et al.* (GYSEL *et al.*, 2016) proposed a semi-automated service identification model according to predefined categories, based mainly on requirement artifacts. Each category was objectively detailed by certain cohesion criteria that the service should present for its endpoints. Through approximation algorithms and use of weights, the found services would be directed to a specific category. That approach can be used to avoid bias or misinterpretation by specifying which endpoint belongs to one or another service. It was unclear how the involvement of business and system experts could help the understanding and validation of those categories, and how it could be ensured that the system decomposition meets the business requirements.

Studies about code refactoring and migration of legacy systems to microservices, which focus on refactoring business class code with the creation of interfaces for emerging microservices, have been proposed (KNOCHE; HASSELBRING, 2018) (LEVCOVITZ; TERRA; VALENTE, 2016). The approach in (KNOCHE; HASSELBRING, 2018) initially asks system users which functionality to migrate at a time and focuses on refactoring only the application source code, not mentioning code implemented in the database. In (LEVCOVITZ; TERRA; VALENTE, 2016), the authors propose a mechanism to subdivide the business model into sub-

models, for which, by refining the database entities, microservices would emerge to meet those subdivisions of the original business model.

Those approaches (KNOCHE; HASSELBRING, 2018) (LEVCOVITZ; TERRA; VALENTE, 2016), which are usually related to extraction of microservices from source code, propose a process that analyses the business requirements of the entire system and groups similar or complementary requirements. The groups are composed of a three part relation: business requirement, code functionality, and data entity. That triad constitutes a candidate to become a microservice implementation. The way that microservices are identified in (KNOCHE; HASSELBRING, 2018) (LEVCOVITZ; TERRA; VALENTE, 2016), considering the business rules and their associated data, is similar to the one proposed in this work. However, while they focus on analysing the application source code, this work investigates the business logic contained in the stored procedures.

A semi-automated approach to microservice discovery and system modularization is described in (BARESI; GARRIGA; RENZIS, 2017), in which a RESTful API is proposed to receive as input a componentized business model that describes the requirement flow, along with its inputs and outputs. The microservices are generated by processing that model. Analogously to our work, the approach in (BARESI; GARRIGA; RENZIS, 2017) relies on the need to understand the requirement flow and to group similar information into distinct microservices. Nonetheless, the authors also do not address business rules that are in database artifacts.

Another difference of those aforementioned approaches and the process proposed in this work is that both suggest a large refactoring in the system that may result in a complete new set of code classes and interfaces (KNOCHE; HASSELBRING, 2018). However, the present work refactors the system at a single point in the source code that directs execution to an artifact in the database.

Usually migrating legacy systems to a microservice architecture involves directly a gradual database refactoring, since there are two main different approaches when dealing with data layer. One of them regards the same database used by the emergent services and adjustments are made in the base in the moment of the microservice integration (YANAGA, 2017). Another one uses the strategy of isolating the section of the database corresponding to the emergent microservice and creating one database for each one, named as Polyglot Persistence (FOWLER, 2011)(LEBERKNIGHTS, 2008). In this work it is used the strategy of maintaining the database shared between the promoted microservices, abstracting all changes of using an approach decoupled from database.

## 4 THE PROPOSED PROCESS

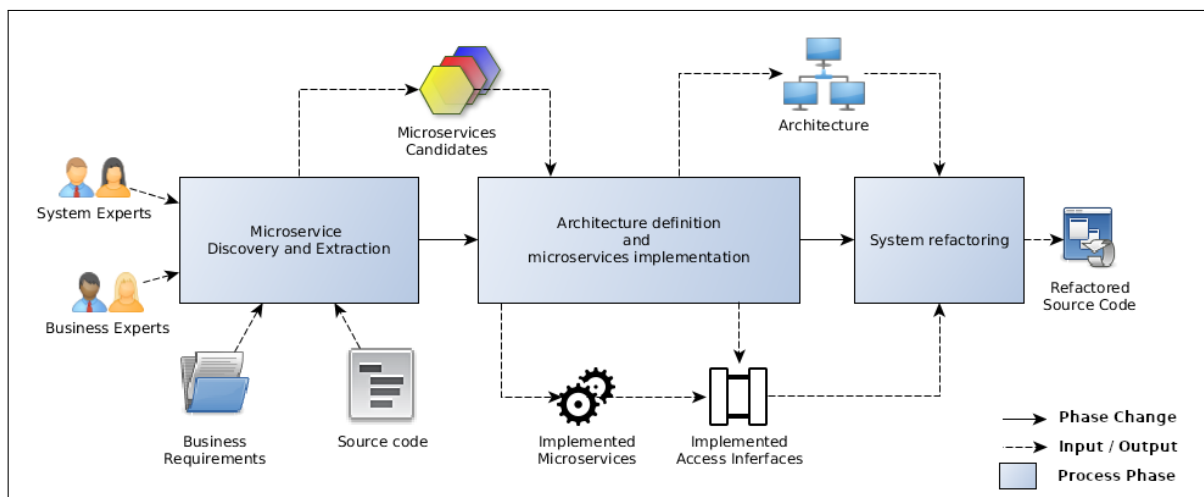
A three-phase process to migrate database-oriented legacy systems is described in this chapter. As a brief overview of the process, the first phase - *Microservice Discovery and Extraction* - presents a way to discover and extract microservices from the database artifacts. The inputs of this phase include the expertise of system specialists, such as business analysts, system analysts and developers, as well as detailed business requirement documents, such as UML artifacts, users stories, prototypes and source code. The output consists of all microservices candidates, which are the inputs to the next phase.

The second phase - *Architecture Definition and Microservices Implementation* - addresses the architectural definition that supports the solution and the creation of the emerged microservices. In this phase, the candidate microservices, which come from the previous phase, are implemented according to a specific microservice architecture. The architecture definition, which depends on the context of the system environment, along with all implemented microservices with their respective access interfaces, are the outputs of this phase.

The last phase - *System Refactoring* - discusses how the system should be refactored to use the new microservices. In this phase, the process user already knows the target architecture, and the interfaces of access to all implemented microservices and uses them as inputs to select the best approach to refactor the system, resulting on a microservice architecture-based system.

The entire process can be seen in Figure 7. Each phase will be detailed in the next sections.

Figure 7 – Process Overview



Source: the author

## 4.1 Microservice discovery and extraction

This section details the proposed process to identify microservice candidates from the analysis of business rules implemented in stored procedures. The process can be divided into three phases: (i) system requirement selection and stored procedure mapping, (ii) microservice candidates discovery, and (iii) microservice candidates validation, as illustrated by Figure 8. The first phase deals with three levels of abstraction: a set of business requirements (higher level), the system features in the source code (intermediate level) that implement the requirements, and the stored procedures (lower level) in which the business rules are processed. The second phase has the goal of analysing the stored procedures to identify the possible microservices and their endpoints, while the last phase aims at evaluating the source code of all microservice candidates in order to refactor them if necessary. Each phase will be describe in detail as follows.

### 4.1.1 System requirement selection and stored procedure mapping

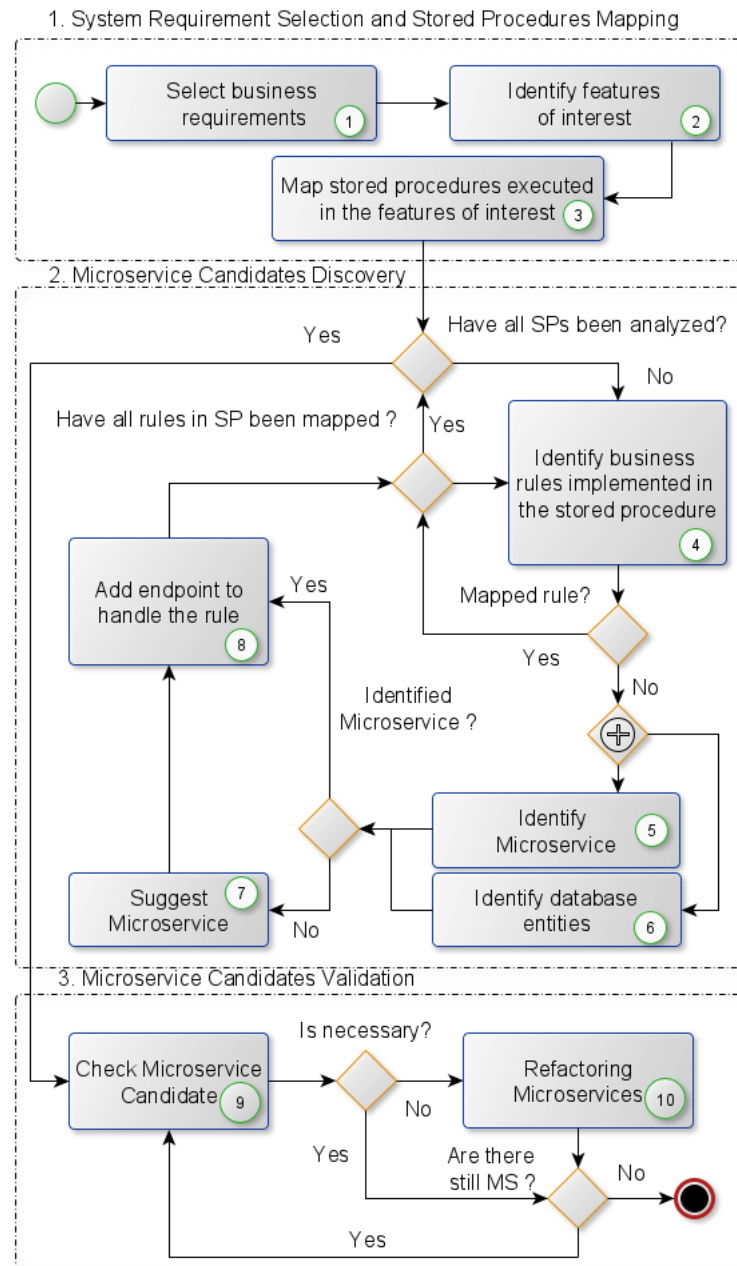
This phase starts by the user selecting a business requirement of the system to analyse (Task 1). It is considered as the user of the process the person who is responsible for conducting the migration. To help the decision on which requirements to select, it is highly recommended that the user asks for help from an expert on the system, who can be a business analyst, software engineer or project manager, among others. Henceforth, (s)he will be referred to as only *system expert*, regardless whether this role is fulfilled by more than one person.

Involving the system expert in this process is fundamental since (s)he guides the business flow and has the knowledge about where each requirement has been implemented. The requirements must be assessed isolatedly, i.e., independently of the others, but the integration of all requirements must be considered to address all business needs. Furthermore, the system expert may indicate the best set of requirements to be considered in the migration to microservices.

When selecting the requirements, it is important that all available information associated to them are taken into account, e.g., UML diagrams, documentation, prototypes, interfaces, business reports, source code, and also the executable system itself. This helps the process' user to map all possible existing business requirements that may be spread across different system features.

The selection of the system requirements can follow different criteria according to the company needs, such as security, business prioritization, requirement preconditions, or system support. Modularization of the system may also be used as a criterion and evaluation

Figure 8 – Detailed process to identify microservice candidates by analysing stored procedures



Source: the author

may be performed module by module, prioritizing the basic modules and so on. In addition, the criteria may depend on other variables such as the interest of some stakeholders involved in the system migration, the development team's knowledge and others that are specific to each company.

In the second task (Task 2 in Figure 8), the user should identify the system features in which the selected requirements are implemented. We refer to them as *features of interest*. One way to do that is to exercise the system under the supervision of the system expert, who can show how to access the specific functionality for each requirement. Another possibility is to

look for that information in the system documentation when it is available and up-to-date.

The next step consists of checking the source code of the features of interest identified in the previous step in order to map the stored procedures that implement the corresponding application business logic (Task 3). Although in this work the focus was only on SPs, the process can be extended in the future to encompass triggers, views and other database artifacts.

As the output of the first phase, a table that brings the selected system requirements, its corresponding system features of interest in the source code and the stored procedures that implement the requirement's business logic is produced. We call this table as *microservice discovery table* (MDT).

#### 4.1.2 Microservice candidates discovery

The second phase of the process investigates each identified stored procedure in order to map all business rules. This will be carried out by an iteration that ends when all business rules of all SPs have been mapped.

It initiates by selecting an SP from the list of SPs as the result of the previous phase. After that, the SP's source code is assessed in order to identify the business rules that it implements (Task 4). Those rules are usually written in sequential execution steps, e.g., a query method followed by an update method, or a nested set of queries, or even an atomic method like a single query or a transaction method (update, insert or delete). One SP can contain several business rules. At this stage, it is recommended to review the documentation available, if its up-to-date, and to count on the system expert's support.

The next step focus on identifying one suggested microservice to implement each business rule in the SP under analysis (Task 5). If a rule does not yet belong to an existing microservice, a new one must be created and the rule is implemented as a new endpoint in this microservice (Task 8). The condition to create or not a new microservice to implement the rule depends on the evaluation from the system expert. Rules must be grouped in the same microservice if they deal with the same business requirement.

Otherwise, if a microservice already exists that treats this type of business requirement, the rule will then be added as a new endpoint to it. The last alternative path is that the rule has been already mapped and there is a microservice that already has an endpoint that implements it. In that case, the process ignores the rule and goes to the next one.

After a business rule is mapped (Task 4) for the first time in the SP, the process has to



identify a candidate microservice to host the mapped rule (Task 5) and all the entities (database tables) that this part of code uses to implement the rule (Task 6). Following the DDD principles (EVANS, 2003), an entity represents a system's domain element, such as customer, product, or student. In this realm, a microservice can be seen as an entity that provides a set of business rules implemented by its endpoints, and that stores data either in its own or in a shared database.

The process searches for all rules in a stored procedure, evaluates all the conditions above and create new microservices, or add new endpoints to existing ones, until all rules are extracted from the procedure. This is repeated until all stored procedures have been analysed.

When this phase is finished, the MDT is augmented with information regarding the microservices, entities, and rules. The process outcome is a set of all microservice candidates, whose endpoints refer to the system business rules. This result is the input to the next step that will validate the identified microservices.

#### **4.1.3 Microservice candidates validation**

The task to check microservice candidates (Task 9) consists of two activities: firstly, to remove all duplicated endpoints and, secondly, to verify whether the microservice represents the business needs. To do that, the system expert analyses whether the microservice is in accordance with the system requirements. In case it is not, then the microservice is deleted.

The next step regards refactoring the microservices by reviewing their endpoints in order to identify whether they really belong to the microservice or whether it is necessary to move them to other microservice. It may be the case that is is necessary to create an appropriate microservice to handle that endpoint (Task 10).

## **4.2 Architecture definition and microservice creation**

When evolving a legacy system to a microservice architecture, it is essential to define the new architecture. A traditional microservice-based architecture is composed of a set of microservices that have their own database, a message broker, a service discovery engine, a load balance and possible an orchestrator to coordinate all services. But migrating a legacy system to this new architectural style is not an easy task, especially when involving data integrity and access.

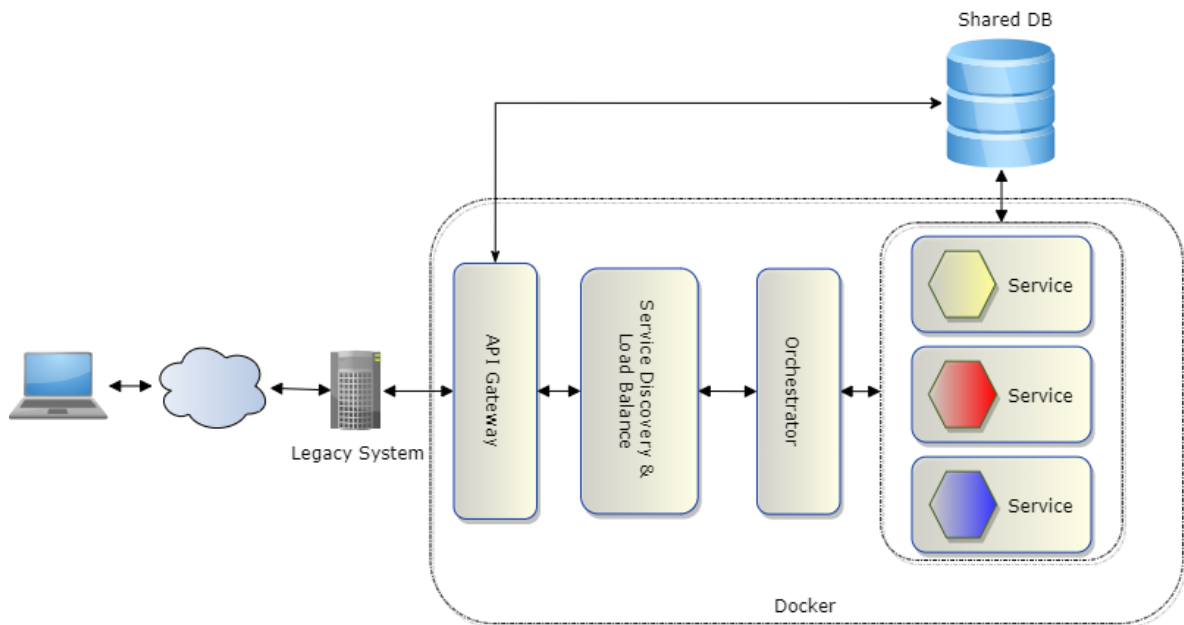
Legacy systems are usually modular, but this does not imply that modules are isolated. Eventually they share the same business rules and dataset. If the migration strategy consists of

migrating each module at a time, it is important that the remain modules work in the same way and the migrated modules work in the new architecture seamless, accessing the same dataset and performing transactions that change the same shared data. In this context, the integrity, synchronization and availability of all data must be ensured.

Isolating databases in each microservice, in this case, will need extra work to synchronize the new database with the legacy one, to ensure full data integrity regardless of which module (migrated or remaining ones) has updated the data. For this, a reasonable approach is to use a hybrid architectural design in which microservices do not have their own database, but share the legacy database. By using this approach, not only the migration effort will be reduced, but also it will demand less refactoring work.

As an architectural framework for addressing the proposed process, this dissertation suggests the architecture presented in Figure 9. This work does not intend to propose a generic architecture that can be used in all cases. It is necessary to evaluate the specific context in all scenarios to propose an appropriate architecture.

Figure 9 – Detailed Proposed Architecture



Source: the author

After defining the microservice architecture, it must be defined how the microservices should be implemented, since this implementation is directly affected by the architecture components. The proposed architecture uses a direct HTTP RESTful access instead of message queue or a message broker for exchanging messages between microservices, as well as the

condition of sharing the same legacy database with the rest of the system.

The proposed implementation is that each business rule mapped from the stored procedures becomes an endpoint in a specific microservice, which can access directly the database or, if necessary, can access another microservices endpoint.

Considering Figure 9, the API Gateway is necessary to route the requests from the legacy system to a specific service. Some architectures bring the API Gateway before the legacy system, switching the requests from the front-end between the system and the services. However this process assumes that the legacy system will be always accessed firstly and the refactoring made settle which service to access. If the request does not call a refactored business requirement, the API Gateway will redirect it to accesses directly the database.

The Service Discovery and Load Balance are common components of a traditional architecture that support microservices. Finally the orchestrator is a service that will coordinate other (micro)services to work together in a predefined sequence of calls to implement a necessary business logic rule.

### **4.3 System refactoring**

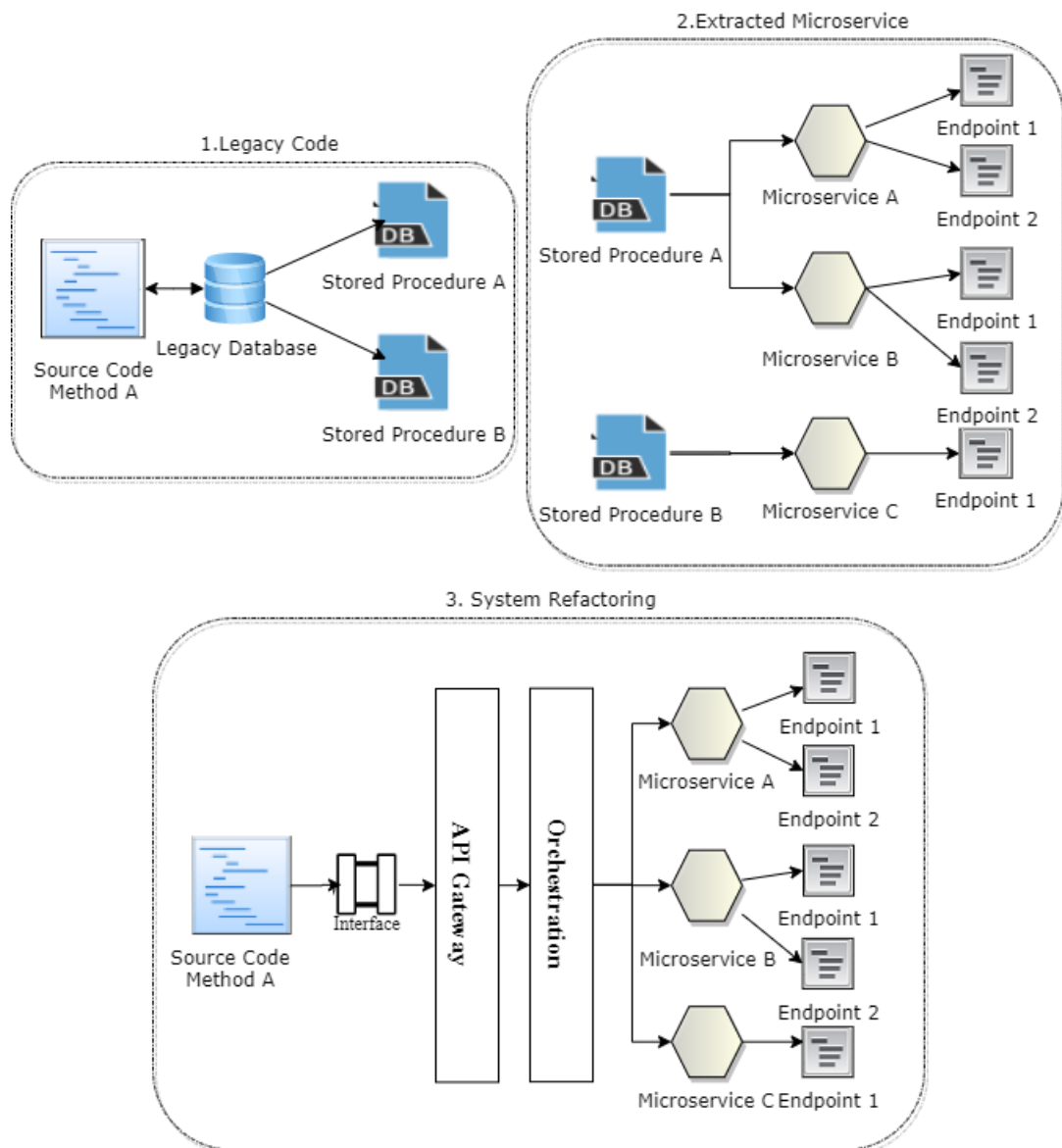
The last process stage encloses the refactoring of the system to use microservices. There are many ways to refactor a system and this work does not want to avoid any of them nor indicate the best approach.

The strategy here is to access the API Gateway by an interface. The interface methods will expose the API Gateway's endpoints that redirect the requests to the orchestrator's endpoints that implement the stored procedures behavior. This approach ensures greater portability as well as low code-coupling, which makes it easy to maintain. All the redirect points mapped in the first process stage will call a specific orchestrator's endpoint through the interface, that override the original stored procedure to ensure business requirements (see Figure 10).

A sequence of endpoints can be called more than one microservice to meet a business requirement. This happens when a stored procedure is mapped to many business rules, all of them should be executed in a specific order to confirm the requirement. Therefore, code refactoring must repeat the same sequence of rules by accessing the endpoints referred to in the orchestrator.

The API Gateway implementation will either route the request to a endpoint in the orchestrator if the procedure was already migrated or redirect the request to the database to access the procedure directly, maintaining the original behavior of the legacy system.

Figure 10 – Proposed Code Refactoring



Source: the author

At the source code method of the legacy system, the refactoring consists on changing the direct stored procedure access to the corresponding method in the interface.

The Strangler Pattern will be used in such a way that, with each new execution of the first phase of the process, a new set of stored procedures will be refactored from the database and implemented in microservices, which will be used by the system, gradually reducing the dependence on direct access to the database.

## 5 PROOF OF CONCEPT

As a proof of concept, we have applied the proposed process to a large scale legacy web system of a Brazilian public IT company. The system's main objective is maintaining and supporting Federal Government infrastructure projects in areas like health, education, security, and transportation. The system allows inserting and searching information on programs, actions, ventures, contracts and covenants; managing the budget for the execution of the projects; and monitoring the execution of actions and ventures. A venture represents a new government infrastructure project in which it is necessary to monitor various factors such as its execution, cost, budget used over time, coordinators, location, among others (e.g a new hospital, federal road, a bridge, a new port).

The system was developed and has been in use since early 2007. It was originally developed in Visual Basic 6 with ASP front-end, but few modules were migrated to ASP.Net using a back-end in C#. The system uses a Microsoft MSSQL Server database, currently in version 2016 and containing about 205,000 records, with 1,521 stored procedures that implement much of the system's business logic.

The system has 97 use cases that enclose all requirements and are organized in five modules: the Registration Module (28 use cases); the Administration Module (24 use cases); the Batch Routine Execution Module (6 use cases); the Commitment Module (10 use cases); and the Monitoring Module (29 use cases) complete the entire system scope.

### 5.1 Legal, ethical and intellectual property issues

To guarantee the intellectual property, ethical, legal and normative issues regarding the custody and disclosure of the content of the legacy system used as the basis for this proof of concept, as well as its source code and the code of the stored procedures analyzed, although the information contained in the database tables used is not from a production environment, all other information are of a confidential nature and must be protected and its disclosure is prohibited in accordance with the rules and regulations of the company responsible for the system. The examples in this work throughout the text have been carefully modified to mischaracterize the original content, without any didactic and proof loss that this work proposes.

## 5.2 Applying the proposed process

### 5.2.1 Phase 1: microservice discovery and extraction

Following the process illustrated by Figure 8, the first step consisted on selecting the business requirements that will be migrated. To do that, we counted on a business analyst who was involved in the development of the system and played the role of the system expert in the process. He gave an overview of the system and the implemented business requirements, such as the main and the alternative business flows, as well as system outputs. The system expert presented all relationships between each system feature and the specific requirement that it implements, where in the documentation the requirement was specified, as well as the code artifact that accesses that functionality.

The first criterion used to select the requirements was to prioritize the core system module due to its relevance to the business. Secondly, we selected the key requirements from the chosen module that add more value to the business (Task 1). This way, the Registration module and, more specifically, four requirements (that traverse the 28 use cases of the module) that act directly in the maintenance of the ventures, were nominated by the business analyst for this proof of concept. The use cases of the Registration module represent 21.6% of the total system's use cases. The selected requirements were: *Identify Venture Project*, *Maintain Venture Project*, *Query Venture Project*, and *Process Venture Project*.

The requirement *Identify Venture Project* covers the creation of the venture, searching of the data of a registered venture, submission of a venture for approval, acceptance by the coordinator, deletion of a venture, reactivation and devolution. The requirement *Maintain Venture Project* is responsible for updating the information necessary for the project monitoring, such as physical and financial execution schedule, pictures of its evolution, budget, among others. The requirement *Query Venture Project* returns the current venture information as well as its historical record information.

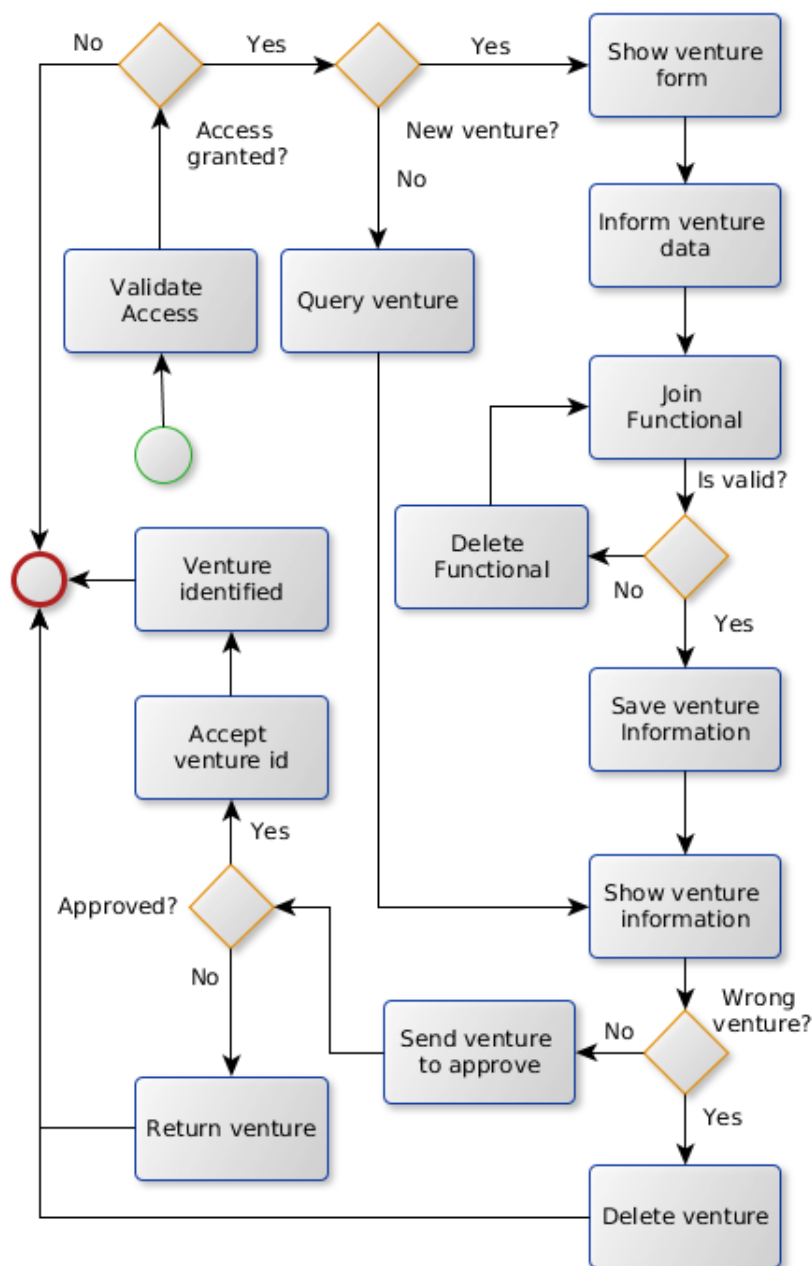
Finally, the requirement *Process Venture Project* compiles venture records and validates the project information by displaying to the manager a list of inconsistencies that should be addressed for the venture. With this, we have performed the first task of the proposed process (Select business requirements).

This phase of the process took a long time, mainly for two reasons: availability and dependence on the system specialist, and to understand and map the flow of business

requirements in the system. It took about 4 months for this phase to be completed, and we have as a result a clipping of the system's business requirements.

Next, it was necessary to identify the system features of interest (Task 2), i.e., the features that implement the requirements. To do that, for each requirement, its complete execution flow, which encloses all features related to the requirement and the possible sequence of system interfaces and their menu items, as well as the necessary information to accomplish the requirement, were detailed by the business analyst.

Figure 11 – Identify Venture requirement flow



Source: the author

After all features of each requirement have been identified, the source code was analysed in order to find out the stored procedures associated to them. At this time, for each activity related to the each requirement flow implementation, it was possible to map the stored procedure call to the database that processes this activity, related to the Task 3 of Figure 8. At the end, 170 stored procedures were mapped to all four requirements. They represent 11.18% of the total stored procedures of the system. Table 1 presents the number of identified SPs for each selected requirement.

Table 1 – Total number of SPs for each selected requirement

ID	Requirement Name	Number of identified SPs
1	Identify Venture Project	36
2	Maintain Venture Project	112
3	Query Venture Project	7
4	Process Venture Project	15

The requirement flow begins by validating the credentials of the user using the “Validate Access” activity, as shown by Figure 11. In this activity, the system checks whether the user has a valid profile registered to access the functionality. At this point the “**sps\_PAC Utils Get Agency Situation Room**”, “**sps\_PAC Utils Get Agency With Action**”, “**sps\_PAC Utils Get User Agency**” and “**sps\_PAC Registration Current Year**” stored procedures were mapped. When the user access has been validated, he can choose to create a new venture or query an identified venture. By opting to query an identified venture, the flow execution will go to the “Query Venture” activity, where the system will use the “**sps\_PAC2 Registration Get Venture Identify**” and “**sps\_PAC2 Registration Get Venture History**” stored procedures, displaying the registered venture identification data. If the user chooses to create a new record, the flow will go to the “Show Venture Form” activity displaying a business registration form. Some information required to perform this registration are returned to the form by executing the stored procedures “**sps\_PAC2 Registration Get Segments**”, “**sps\_PAC Utils Get Sectors**”, “**sps\_PAC Registration Get Budget Units By Agency**”, “**sps\_PAC2 Registration Get Actions PAC**”, “**sps\_PAC Utils Get Ventures In Edition**”, “**sps\_PAC2 Registration Get Action Detail**”, “**sps\_PAC2 Registration Get Authorization Amount**” and “**sps\_PAC2 Registration Get Functional Last Year**”.

The activity “Join Functional” executes the SP “**sps\_PAC2 Registration Join Functional**”, and the related functional activity “Delete Functional” executes the SPs “**sps\_PAC2 Registration Get Functionals**” and “**sps\_PAC2 Registration Delete Functional**”. The acti-



vity “Save Venture Information” runs the SPs “**spa\_PAC2 Registration Save Venture**” and “**sps\_PAC2 Registration Get Venture Id**” .

This excerpt makes it possible to view the distribution of stored procedures throughout the execution of the *Identify Venture Project* requirement flow.

As the result of this phase, Figure 13 shows an excerpt of the MDT for the requirement *Identify Venture Project* containing 10 mapped stored procedures. At this time, only columns “Stored Procedure Name” and “Identified Business Rules” were filled in.

Now the phase 2 of the process begins by iterating through each stored procedure and performing a thorough code analysis (Task 4). Considering the requirement *Identify Venture Project*, Figure 12 represents the analysis of the stored procedure “**spa\_PAC2 Registration Join Functional**” source code, where the selected code indicates a specific business rule according to the business analyst’s indication (Task 5). The rules 01, 02 and 03, depicted in Figure 12, were mapped as “Check if the functional is joined to the venture”, “Join / Update functional in the venture”, and “Return venture functional Id”, respectively, in the column “Identified Business Rules” of the MDP shown in Figure 13). In addition to the implemented rules, the indication of the parameters used in the stored procedures can help the creation of the endpoints that will be implemented in the microservices.

Continuing the process, tasks 7 and 8 (see Figure 8) were performed for each identified business rule in each stored procedure, verifying whether the rule had already been mapped, otherwise checking whether the new rule belongs to an existing microservice and, lastly, identifying a new microservice candidate. Those information can be seen in columns “New Rule?”, “Microservice Name”, and “Situation” of the microservice discovery table. Figure 13 shows that information for some SPs of the requirement *Identify Venture Project*.

The criterion for grouping the rules in a given microservice was primarily the main entity (database table) affected by the business rule, e.g., if the main entity addressed in a rule is “PACVenture”, then that rule was directed to the Venture microservice. For this reason, the entities that are part of the mapped rules are identified (Task 6). Whenever a rule was identified in a stored procedure, all previous rows of the MDT were checked to see if the rule was a new one, if the microservice was new (Task 7), or if that rule belonged to an existing microservice (Task 8), as well as if neither the rule nor the microservice were new. Figure 13 shows the “Entities” column that lists each entity used to implement the business rule in the stored procedure (Task 6).

We also noticed that some rules were repeated in various stored procedures and

Figure 12 – Discovering Rules in the Source Code  
of the SP “spa\_PAC2RegistrationJoinFunctional”

```
CREATE PROCEDURE [dbo].[spa_PAC2RegistrationJoinFunctional]
    @VentId int, @VentYear int, @MOMId int,
    @ESFId char(2), @UNIId char(5), @FUNId char(2),
    @SFUID char(3), @PRGId char(4), @ACAId char(4),
    @LOCId char(4), @SACId char(4), @PRGAno int,
    @Tipo int, @Rap int, @Unica int, @UNITpoId char(1),
    @FALUsuarioAlt int, @FALSisSiglaAlt char(15)
as
    if not exists(select 1 from pacfunctional where
        VentId = @VentId and
        VentYear = @VentYear and
        MOMId = @MOMId and esfId = @esfId and
        uniId = @uniId and funId = @funId and
        sfuId = @sfuId and prgId = @prgId and
        acaId = @acaId and locId = @locId and
        sacId = @sacId and falsNRap = @Rap)
    begin
        insert into pacfunctional (VentId, VentYear, MOMId,
            esfId, uniId, funId, sfuId, prgId, acaId, locId, sacId,
            PRGAno, FALAcaoPrincipal, FALSNRap, FALSNUnica, UNITpoId,
            FALUsuarioAlt, ALSisSiglaAlt, FALDataAlt)
        values(@VentId, @VentYear, @MOMId, @ESFId,
            @UNIId, @FUNId, @SFUID, @PRGId, @ACAId,
            @LOCId, @SACId, @PRGAno, @Tipo, @RAP, @Unica,
            @UNITpoId, @FALUsuarioAlt, @FALSisSiglaAlt,
            getdate())
    end
    else
    begin
        update pacfunctional
        set FALAcaoPrincipal = @Tipo,
            FALSNUnica = @Unica,
            FALUsuarioAlt = FALUsuarioAlt,
            FALSisSiglaAlt = @FALSisSiglaAlt,
            FALDataAlt = getdate(),
            FALSNEExcluir=0
        where
            VentId = @VentId and VentYear = @VentYear and
            MOMId = @MOMId and esfId = @esfId and
            uniId = @uniId and esfId = @esfId and
            sfuId = @sfuId and prgId = @prgId and
            acaId = @acaId and locId = @locId and
            sacId = @sacId and falsNRap = @Rap
    end

    select falId as falId
    from PACfunctional as functional
    where
        VentId = @VentId and
        VentYear = @VentYear and
        MOMId = @MOMId and esfId = @esfId and
        uniId = @uniId and funId = @funId and
        sfuId = @sfuId and prgId = @prgId and
        acaId = @acaId and locId = @locId and
        sacId = @sacId and falsNRap = @Rap
```

↑ PARAMETERS

← RULE 01

↑ RULE 02 ↓

← RULE 03

Source: the author

provided orthogonal behaviour to the system business rules. This may indicate that those rules are possibly part of control mechanisms that should be checked at various times during business flow executions, such as user authentication, domain information, general system settings, among others. Our solution to deal with that was to create a generic microservice called “Utils” and to insert those crosscutting rules as its endpoints.

Figure 13 – Excerpt of the Microservice Discovery Table for requirement *Identify Venture Project*

	Stored Procedure Name	Identified Business Rules	New Rule?	Entities	Microservice Name	Situation
1	sps_PACUtilsGetAgencySituationRoom	Return agencies from a specific situation	Yes	PacSituationRoom, PacSectorRoom, PacSituationRoomParticipants, PacSegment, PacVenture, Unit, Agency	SituationRoom	New Microservice
2	sps_PACUtilsGetAgencyWithAction					Not mapped
3	sps_PACUtilsGetUserAgency	Return current exercise	Yes	PACYearExercise	Utils	New Microservice
		Return user access level	Yes	AccessLevel	AccessLevel	New Microservice
		Return user agency with access 1	Yes	Agency, UserLevel, Unit, Action, ActionProgramGroup	Agency	New Microservice
		Return user agency with a specific access	Yes	UserLevel, Agency	Agency	New rule + Existing MS
4	sps_PACRegistrationCurrentYear	Return current exercise	No	PACYearExercise	Utils	Existing rule
5	sps_PAC2RegistrationGetVentureIdentify	Return venture status description	Yes	PACStatus	Venture	New Microservice
		Return venture state	Yes	PACVentureState	Venture	New rule + Existing MS
		Check if venture has a valid moment	Yes	PACVenture	Venture	New rule + Existing MS
		Return a venture to identify	Yes	PACVenture, Unit, Agency, PacSegment, PacSector, PacVentureTypeWallet, PacFunctional, Action, Region, PacAgencyMnemonic	Venture	New rule + Existing MS
6	sps_PAC2RegistrationGetSegments	Return segments list	Yes	PACSegment	Utils	New rule + Existing MS
7	sps_PAC2RegistrationGetFunctionalLastYear	Return the last year venture functionals	Yes	PACFunctional, Action, Region	Functional	New Microservice
8	sps_PAC2RegistrationGetVentureHistory	Return the change history of a venture	Yes	PACLog, PACStatus, AccessLevel, User	Venture	New rule + Existing MS
9	sps_PACRegistrationGetBudgetUnitsByAgency	Return units by type and agency	Yes	Unit	Utils	New rule + Existing MS
10	spa_PAC2RegistrationJoinFunctional	Check if the functional is joined to the venture	Yes	PACFunctional	Functional	New rule + Existing MS
		Join / Update functional in the venture	Yes	PACFunctional	Functional	New rule + Existing MS
		Return if venture functional exists	Yes	PACFunctional	Functional	New rule + Existing MS

Source: the author

At the end of phase 2, 362 business rules were identified across all 170 analysed stored procedures that implement the four selected requirements, resulting in 15 microservice candidates.

After the analysis of all stored procedures, the identification of all possible microservice candidates and their business rules, the last phase of the process aimed at validating the identified microservices in order to verify whether they were really necessary or whether they needed to be refactored by removing duplicated business rules or moving some business rules to a most appropriate microservice (task 9). From the total number of found business rules, 155 rules (42,82%) were identified as duplicated, thus indicating a lack of code reuse and increasing the complexity of the system maintenance. Furthermore, two microservices were considered unnecessary, since they contained only one business rule each. Hence, those microservices have been removed and the rules moved to other microservices (Task 10). At the end of the whole process, 13 microservice candidates were proposed to the migration.

Figure 14 summarises the first findings of this proof of concept. Note (\*) that 11 stored procedures could not be mapped for two reasons: they either used functions that were not accessible at the time of evaluation or had high complex code that needed a system analyst/developer intervention. Literal rule duplication (\*\*) means that 42 rules implemented in the 170 evaluated procedures code were exactly duplicated in some other procedure (\*\*\*).

Figure 14 – Proof of Concept Findings

Findings	Values
Total System Uses Cases	97
Total System Stored Procedures	1521
Evaluated Use Cases	28
% in relation to the total use cases	28,87%
Total Stored Procedures evaluated	170
% in relation to the total System Sps	11,18%
Candidate Microservices	15
Reviewed Microservices	13
Mapped Rules (procedures evaluated)	362
New Rules (procedures evaluated)	207
Duplicated Rules (procedures evaluated)	155
% duplicated rules	42,82%
Not mapped procedures *	11
Implemented stored procedures in ms	32
% in stored procedures evaluated	18,82%
Implemented rules (ms endpoints)	71
% new rules implemented	34,30%
Duplicated implemented rules (ms endpoints)	55
% duplicated rules implemented	35,48%
Literal rule duplication **	42
Procedures with literal rule duplication ***	137
% procedures with literal rule duplication	9,01%

Source: the author

## 5.2.2 Phase 2: architecture definition and microservice creation

Here it is described, in specific subsections, those two activities that compose the phase 2 of the process.

### 5.2.2.1 Implementing microservices candidates

The next phase of this proof of concept was to implement the microservices mapped during the discovery and extraction phase and to construct the new system architecture. For this phase the following tools were used:

- IDE - Spring Tool Suite 4, version 4.6.2 <sup>1</sup>;
- Database - MSSQL Server 2014;
- HTTP REST Client - Postman v7.29.1.

The first step was to restore the backup of the legacy system development database on the SQL Server 2014 to access the stored procedures. The next step was to create a Java project for a Spring Boot <sup>2</sup> framework application, version 2.3.1, in order to implement each discovered microservice. The service will access the database through the Hibernate framework, version 5.3.4. Spring Boot was chosen because it is a tool that facilitates the configuration and

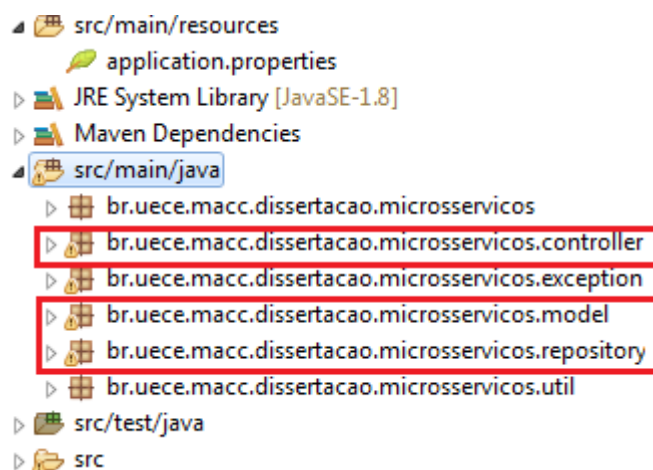
<sup>1</sup> <https://spring.io/tools>

<sup>2</sup> <https://spring.io/projects/spring-boot>

publication of applications that use the Spring ecosystem. It allows you to choose the modules you want for your application (e.g. WEB, Template, Persistence, Security) and has continuous integration configuration facilities. Another great facility is that Spring Boot brings an instance of Tomcat as an “embedded” web server, which removes the need to configure the web server where the application will be published.

The package structure of the project was designed containing three main packages: Model, Repository and Controller (see Figure 15). The Model package contains the classes that would be mapped by Hibernate on the database. The Repository package stores the classes that implement the business rules extracted from the stored procedures. Finally, the controller package exposes the microservice endpoints.

Figure 15 – Microservice Project Packages



Source: the author

Using the MDT (see Figure 13), the project was named with the microservice identification and a repository class was created containing the methods, named with mnemonics that represent the rules identified for a specific stored procedure for that microservice. In line number 3 of the MDT, for example, there is a microservice named “Utils”, and a business rule identified as “Return Current Exercise”. As an implementation of this line, a new Spring Boot project called “Utils” (the same microservice name) will be created, and a repository class will be created in this project containing a method to represent the identified business rule, called “returnCurrentExercise”. The proposed method should execute the part of the stored procedure code identified as this business rule, in this example, the stored procedure “sps\_PACUtilsGetUserAgency”. After the repository class and method were created, a controller class was created to expose that method as a microservice endpoint.

After created, the microservice endpoint was tested using the Postman tool by passing all necessary parameters for execution. The result was compared with the execution of the stored procedure code directly on the database management tool (SQL Server 2014 Management Studio), using the same parameters. If the two results are the same, then it is considered that the test succeeds.

For each row in the MDT, this process was repeated for creating either a new microservice or a method in an existing one, incrementally. For this proof of concept, the MDT has not been completely revisited. Only stored procedures belonging to the *Identify Venture* requirement were considered because the process would become repetitive, not adding more value to what was intended, but without loss of generality.

During the implementation process, it was possible to validate whether the microservice candidate was really needed or whether the endpoint should belong to another microservice. Hence, the MDT was updated when necessary.

For the analysed requirement, 32 stored procedures out of 170 mapped (18.8%) in the MDT were evaluated. From those, 11 microservices were created, although at this moment, without all the mapped business rules listed in the MDT, only those that were necessary to implements the requirement. During the implementation, seven methods belonging to another microservice were identified, as well as three unnecessary methods. 71 methods were created, referring new business rules, distributed among the 11 microservices, representing 34.3% of all rules mapped in the MDT (see Figure 14). 55 duplicate or similar methods were found among the 155 methods for all stored procedures, representing 35.48% of the duplicated rules.

After creating all the necessary microservice projects to cover all identified rules, a microservice orchestration project was created. This service aims at coordinating the microservices to implement the entire context of execution of the stored procedure, thus realizing the business requirement implemented by the procedure. The orchestrator used the same project structure of the other microservices, with only one difference in its controller class. The exposed endpoints had the same procedure names and also the same input parameters. Likewise, the orchestrator's endpoints must return the same result as the stored procedure. Postman was used to execute the endpoint and to validate the result against the return of the stored procedure.

Finally, another service called "LegacySystem" was created to directly execute the stored procedure, according to the original behavior of the legacy system. This microservice has only one endpoint that receives a JSON as a parameter that contains the information of which stored procedure should be executed, as well as all the input and output parameters. The

service will be used as a bypass through the API Gateway. For example, if the stored procedure has already had its rules implemented in microservices, API Gateway will call the orchestrator service, which in turn will call the necessary microservices in a coordinated way to implement the execution of the rules extracted from the procedure. Otherwise, the API Gateway will simply call the LegacySystem service informing the input and output parameters, and which stored procedure will be executed in the database, simulating the direct execution of the procedure by the legacy system.

The last microservice project created was the API Gateway. This service is responsible for forwarding all requests from the legacy system, sometimes routed to the orchestrator, to an endpoint that represents a refactored stored procedure or routed to the microservice that works as a bypass to the stored procedure.

#### 5.2.2.2 **Setting up microservice infrastructure**

The Spring Tool Suite IDE creates a container with an embedded TomCat for each Spring Boot application that runs the implemented microservice project in a specific port of the web server. Thus, each microservice has its own web server and works independently.

A project using Eureka was created as a service registry. Eureka is an open source service originally developed by Netflix for resilient middle-tier load balancing and fault tolerance. That new service has as clients all the implemented microservices, which should inform in their property files the Eureka service for which they have to connect. All microservices should be identified by a unique Id and, after connected to the Eureka server, they can be found by their own Id, without specifying an IP address. The Eureka server can also be configured to load balance microservices and ensure service availability in case of failure.

According to the architecture proposed in this work, an API Gateway service was defined as previously mentioned. That service extends another open source service also developed by Netflix, the Zuul service, to work as a proxy of HTTP requests. The application receives an annotation as Zuul Proxy and configures in its properties all possible routes, depending on the original request. For the purpose of this proof of concept, only two routes were defined, the first for the orchestrator and the second for the service that performs the direct execution of the stored procedure in the database. The API Gateway is also a client of the Eureka server, so no matter the physical location of the other services is, it just uses the service's Id to locate them.

For legal reasons, in this work, only the interface between the legacy system and

the API Gateway was not implemented, because it would not be possible to refactor the legacy system. In this case, the Postman <sup>3</sup> tool was used to simulate the behavior of the interface when requesting an endpoint from the orchestrator or API Gateway.

### 5.2.3 Phase 3: system refactoring

During the implementation of the microservices, the MDT was revised and the indication of code duplication of the same business rule, previously done manually, was confirmed in many stored procedures, when it was verified that the code extracted for the microservice was the same or similar in other evaluated procedures. That directly implies an increase in the complexity and maintenance difficulty of the legacy system, since the same code is spread throughout several stored procedures.

To confirm the hypothesis that the use of microservices implies in improving the maintainability of the system, an analysis was carried out on the pieces of code that were isolated for each business rule to be implemented in each microservice. As mentioned earlier in this work, when creating the MDT (see Figure 13), manually, similarity was found between code snippets from different stored procedures that implemented the same business rule, considering the 170 stored stored procedures evaluated. In the MDT, these rules were identified as existing rules and counted as duplicate rules (see Figure 14).

Extending this verification, a Java program was created that scanned all the metadata of the stored procedures present in the database, totaling 1521 business stored procedures. The program extracted the text of the stored procedure and performed a treatment to remove line breaks, tabs and spaces, transforming the total text of the procedure into a single line, separating reserved words, operators, fields, variables and parameters by characters of type "|". For each procedure that had its text treated, a line was stored in a recent created entity table (T1) in the database (see Figure 16) that contained the name of the procedure and its treated text as attributes.

The same process was performed for each piece of code referring to the extraction of business rules for each procedure of the 170 evaluated, which were not counted as duplicate rules in the previous manual analysis. The result was stored in another new entity table (T2) in the database (see Figure 17) where for each new procedure rule a line was created containing the procedure name and the rule text as attributes, with the same treatment that was defined for the procedure code in the previous table. The difference is that in T1 there is only one line for each

---

<sup>3</sup> <https://www.postman.com/>





rules, or where there would be duplication of code in the procedures, but rather to point out in a striking and unequivocal way that duplication exists and contributes to increase the complexity of maintaining the code, and on the other hand, showing ease of maintenance when the code is encapsulated in a single microservice.

## 6 DISCUSSION AND LESSONS LEARNED

After the application of the process in a real large scale system, some lessons could be learnt. They are discussed as follows.

Firstly, there is a real need for the system expert support. Understanding the business rules and identifying them in the system is not trivial, even when quality documentation is available. Many misunderstandings can happen without this monitoring. In addition, it is essential to fill the MDT carefully, otherwise there can be both lack and repetition of information, which may generate wrong results and reworking.

Another lesson learned was the need for a local environment for system simulation and source code access. When dealing with a real system with data in production, sometimes accessing the system is not simple and requires prior authorization. As it is a legacy system that uses technologies such as development frameworks and old databases, it was very challenging to set up a functional environment that mirrors the production environment.

A third lesson consisted on extracting a clipping of the system for this proof of concept. Usually legacy systems are very extensive and complex, and selecting a part that represents the system well is not always a simple task. That decision may be alleviated if a company already knows which features of the systems should be migrated. In addition, the proposed process can be carried out several times in order to migrate the system functionalities incrementally, thus following a Strangler pattern (NEWMAN, 2019).

How the data will be accessed by the new microservices influences the refactoring steps necessary to implement and integrate the identified microservices into the system. Usually migrating legacy systems to a microservice architecture involves a gradual database refactoring, since there are two main different approaches when dealing with data layer (TAIBI; LENAR-DUZZI; PAHL, 2018). One of them regards the same database used by the emergent services and adjustments are made in the base at the moment of the microservice integration (YANAGA, 2017). Another one uses the strategy of isolating the section of the database corresponding to the emergent microservice and creating one database for each microservice, or having a variety of data storage technologies for different types of data, named as Polyglot Persistence (FOWLER, 2011).

Due to the complexity of the system used in the proof of concept, and the fact that it is still running and in use, the most appropriate strategy would be maintaining a shared database between the identified microservices and the rest of the legacy system.

The microservice code implementation is not a trivial task, because it is necessary to reopen the stored procedure one by one, to extract a slice of code from them, to make sure the code is working separately and in accordance with the requirements of the business rules and, finally, to find parameters that make sense to execute the SQL statement with the data available. Furthermore, after that, it is also necessary to reassembly all the stored procedures steps into the orchestrator endpoint.

It was not the objective of this work to evaluate the performance of the functionalities after its implementation using microservices, but, by observing the response time of the functionality's execution directly in the database, and comparing it with the same execution using microservices, it was noticed there was a delay in the response. It was expected that by adding more layers of processing between the beginning and the end of the execution of the functionality the performance would be degraded, however it is also expected that this problem will be compensated when scaling the functionality, as well as in the future using a database for each microservice.

At last, but not least, although the proposed approach helped to identify microservice candidates, the whole process became very tiring and time consuming since it was conducted manually only by one person with the support of the system expert. Thus, automating the process is necessary to tackle those problems, specially code extraction and microservices implementation, and is intended as future work.

## 7 CONCLUSION

This work presented a process that aids the migration of legacy systems to a more recent and steadily-growing architecture based on microservices. Different from other approaches, this process focuses on identifying microservice candidates from database artifacts, more precisely stored procedures, rather than from source code. This can benefit several companies that still use and maintain legacy applications developed during the 1980 and 1990's decades, when due to the robustness of the RDBMS, many developers moved the system business rules to the database with the objective to improve performance.

The proposed process was carefully detailed, specifying the entrances and exits of each phase, as well as all internal steps, in order to follow the evolution of information processing throughout its execution.

The work has been validated by a proof of concept that applied the proposed process to a real large scale system. Four main requirements, which addressed 28 system use cases of a core module of the system, were investigated resulting in 170 analysed stored procedures and 13 identified microservice candidates. Furthermore, we could also detect many duplicated pieces of code, which could be turning the system maintenance more complex.

After that, the candidates for microservices were implemented, evolving the MDT entries and evaluating all the results of the process. In addition, an architecture model oriented to microservices was created to support this proof of concept.

### 7.1 Limitation

Some limitations were identified during the development of this work. The first that can be listed is the great dependence at certain times on a specialist in the system. Access to this professional is not always possible, which can lead to delays in some definitions and make research time-consuming.

As a second limitation, the lack of automated tools for extracting the rules and the search for similar rules in other stored procedures. The entire verification process, carried out in a purely visual way, is prone to errors in interpretation and verification, especially in long stretches of code. In the same way, another limitation would be the need for a tool to convert the extracted rules into microservices.

The research itself was self-limiting by excluding other artifacts from the database that could execute business rules, such as triggers and functions. However, the process can be

easily extended to these other artifacts.

It can also be listed as a limitation the lack of access and permission to analyze and refactor code from proprietary systems, as well as access and manipulation of legacy databases. Usually, this information and codes are confidential and governed by intellectual property rules subject to criminal and administrative sanctions in a corporate environment. Thus, the researcher's study universe can be restricted to a few systems.

Using Postman as a client application was a limitation, as it was not possible to see how the system would need to be refactored to use the interface (API Gateway) to access microservices.

Also as a limitation, it is possible to list the need at the first moment to use a shared database, contrary to what is indicated for an microservice-oriented architecture. This limitation at the moment, is due to the need to parallel the legacy system with microservices and to avoid increasing the complexity of the process, such as the introduction of complex mechanisms to guarantee the integrity of the information. However, strangling all the functionality of the legacy system in the future, it would be good practice to also refactor and isolate the database for each microservice.

A final limitation that can be pointed out is the practical feasibility of applying the work in a corporate environment that does not have an infrastructure that supports the microservice architecture.

## 7.2 Publication

Part of the research developed in this work resulted in a publication entitled **“Towards Identifying Microservice Candidates from Business Rules Implemented in Stored Procedures”**, in the IEEE International Conference on Software Architecture 2020 (ICSA 2020)<sup>1</sup>. In the article, only the process of searching and extracting microservices was presented, which here in this work was continued by the phase of defining the architecture, applying the proof of concept and analyzing the results.

Although not related to the subject of this work, during the academic period, another article published at the **XVIII Brazilian Symposium on Human Factors in Computer Systems (IHC 2019)**<sup>2</sup>, entitled **“A Landscape of the Adoption of Empirical Evaluations in the Brazilian Symposium on Human Factors in Computing Systems”**, it was important for the

<sup>1</sup> <http://icsa-conferences.org/2020/index.html>

<sup>2</sup> <http://ihc2019.ufes.br/>

growth as a researcher of the author, mainly because it is a work that evaluated production techniques and structuring of empirical studies, knowledge that was applied in the structure of this dissertation work.

### **7.3 Future work**

As future work, it would be interesting to finalize the application of the process to the other system modules and to analyze the final results. In addition, it is intended to apply the proposed process to other legacy systems and compare the results among the different systems. Considering that other users rather than this work's author could perform the process, then it could allow the evaluation of the process from the perspective of the user, thus identifying possible new improvement points.

Another interesting research consists of extending the process to discovery microservices from other database artifacts such as triggers, functions and views, and also including code artifacts like test cases. Finally, it would be interesting to use machine learning techniques to support process automation and black box / white box tests to validate the extracted microservices.

## REFERENCES

- BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: Migration to a cloud-native architecture. **IEEE Softw.**, IEEE Computer Society Press, v. 33, n. 3, p. 42–52, maio 2016.
- BARESI, L.; GARRIGA, M.; RENZIS, A. D. Microservices identification through interface analysis. In: PAOLI, F. D.; SCHULTE, S.; JOHNSEN, E. B. (Ed.). **Service-Oriented and Cloud Computing**. Cham: Springer International Publishing, 2017. p. 19–33. ISBN 978-3-319-67262-5.
- BASS, L.; WEBER, I.; ZHU, L. **DevOps: A Software Architect's Perspective**. New York: Addison-Wesley, 2015. ISBN 978-0-13-404984-7. Disponível em: <<http://my.safaribooksonline.com/9780134049847>>. Acesso em: 09 nov. 2020.
- Bucchiarone, A.; Dragoni, N.; Dustdar, S.; Larsen, S. T.; Mazzara, M. From monolithic to microservices: An experience report from the banking domain. **IEEE Software**, v. 35, n. 3, p. 50–55, May 2018.
- BUSCHMANN, F.; HENNEY, K. **Pattern-oriented software architecture: A System of Patterns**. [S.l.]: Wiley, 1996. v. 1. ISBN 978-0471958697.
- CONWAY, M. How do committees invent? F. D. Thompson Publications, Inc, 1968. Disponível em: <[http://www.melconway.com/Home/Conways\\_Law.html](http://www.melconway.com/Home/Conways_Law.html)>. Acesso em: 09 nov. 2020.
- ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems**. 7th. ed. [S.l.]: Pearson, 2015. ISBN 0133970779, 9780133970777.
- EVANS. **Domain-Driven Design: Tacking Complexity In the Heart of Software**. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321125215.
- FONG, J.; HUI, R. Application of middleware in the three tier client/server database design methodology. **J. Braz. Comp. Soc.**, v. 6, p. 50–64, 07 1999.
- FOWLER, M. Stranglerfigapplication. 2004. Disponível em: <<https://martinfowler.com/bliki/StranglerFigApplication.html>>. Acesso em: 09 nov. 2020.
- FOWLER, M. Polyglotpersistence. 2011. Disponível em: <<https://martinfowler.com/bliki/PolyglotPersistence.html>>. Acesso em: 09 nov. 2020.
- FRITZSCH, J.; BOGNER, J.; WAGNER, S.; ZIMMERMANN, A. Microservices migration in industry: Intentions, strategies, and challenges. In: . [S.l.: s.n.], 2019.
- GOUIGOUX, J.-P.; TAMZALIT, D. “functional-first” recommendations for beneficial microservices migration and integration lessons learned from an industrial experience. In: . [S.l.: s.n.], 2019. p. 182–186.
- GYSEL, M.; KÖLBENER, L.; GIERSCHE, W.; ZIMMERMANN, O. Service cutter: A systematic approach to service decomposition. In: AIELLO, M.; JOHNSEN, E. B.; DUSTDAR, S.; GEORGIEVSKI, I. (Ed.). **Service-Oriented and Cloud Computing**. Cham: Springer International Publishing, 2016. p. 185–200. ISBN 978-3-319-44482-6.
- JAMSHIDI, P.; PAHL, C.; MENDONÇA, N.; LEWIS, J.; TILKOV, S. Microservices: The journey so far and challenges ahead. **IEEE Software**, v. 35, p. 24–35, 05 2018.



KNOCHE, H.; HASSELBRING, W. Using microservices for legacy software modernization. **IEEE Software**, v. 35, n. 3, p. 44–49, 05 2018.

LAHIO F. LAUX, I. P. M.; DERVOS, D. **SQL Stored Routines: Procedural Extensions of SQL and External Routines in Transaction Context**. 2017. Disponível em: <[http://www.dbtechnet.org/papers/SQL\\_StoredRoutines.pdf](http://www.dbtechnet.org/papers/SQL_StoredRoutines.pdf)>. Acesso em: 09 nov. 2020.

LEBERKNIGHTS, S. Polyglot persistence. 2008. Disponível em: <[http://www.sleberknight.com/blog/sleberkn/entry/polyglot\\_persistence](http://www.sleberknight.com/blog/sleberkn/entry/polyglot_persistence)>. Acesso em: 09 nov. 2020.

LEVCOVITZ, A.; TERRA, R.; VALENTE, M. T. Towards a technique for extracting microservices from monolithic enterprise systems. **CoRR**, abs/1605.03175, 2016. Disponível em: <<http://arxiv.org/abs/1605.03175>>. Acesso em: 09 nov. 2020.

LEWIS, J.; FOWLER, M. Microservices. 2014. Disponível em: <<http://martinfowler.com/articles/microservices.html>>. Acesso em: 09 nov. 2020.

LINGER, R.; MILLS, H.; WITT, B. **Structured Programming: Theory and Practice**. [S.l.: s.n.], 1979. ISBN 978-0-201-14461-1.

MELTON, J. **Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features**. USA: Elsevier Science Inc., 2002. ISBN 1558606777.

MILLS, H. Structured programming: Retrospect and prospect. **Software, IEEE**, v. 3, p. 58 – 66, 12 1986.

NADAREISHVILI, I.; MITRA, R.; MCLARTY, M.; AMUNDSEN, M. **Microservice Architecture: Aligning Principles, Practices, and Culture**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2016. ISBN 1491956259, 9781491956250.

NAVEA, P. C.; ASTUDILLO, H.; HILLIARD, R.; COLLADO, M. Assessing migration of a 20-year-old system to a micro-service platform using atom. In: . [S.l.: s.n.], 2019. p. 174–181.

NEWMAN, S. **Building Microservices**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2015. ISBN 1491950358, 9781491950357.

NEWMAN, S. **Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith**. O'Reilly Media, Incorporated, 2019. ISBN 9781492047841. Disponível em: <<https://books.google.com.br/books?id=iul3wQEACAAJ>>. Acesso em: 09 nov. 2020.

NUNES, L.; SANTOS, N.; SILVA, A. R. From a monolith to a microservices architecture: An approach based on transactional contexts. In: BURES, T.; DUCHIEN, L.; INVERARDI, P. (Ed.). **Software Architecture**. Cham: Springer International Publishing, 2019. p. 37–52. ISBN 978-3-030-29983-5.

OROUMCHIAN, F. An effective strategy for legacy systems evolution. **Journal of Software Maintenance**, v. 15, p. 325–344, 09 2003.

PAHL, C.; JAMSHIDI, P. Microservices: A systematic mapping study. In: INSTICC. **Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER**. [S.l.]: SciTePress, 2016. p. 137–146. ISBN 978-989-758-182-3.

PAHL, C.; JAMSHIDI, P.; ZIMMERMANN, O. Architectural principles for cloud software. **ACM Transactions on Internet Technology**, v. 18, 06 2017.

RATIONAL. **Rational Unified Process - Best Practices for Software Development Teams**. 2011. Disponível em: <[https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251\\_bestpractices\\_TP026B.pdf](https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf)>. Acesso em: 09 nov. 2020.

SARKAR, S.; RAMACHANDRAN, S.; KUMAR, G.; IYENGAR, M.; RANGARAJAN, K.; SIVAGNANAM, S. Modularization of a large-scale business application: A case study. **IEEE Software**, v. 26, p. 28–35, March 2009.

TAIBI, D.; LENARDUZZI, V.; PAHL, C. Architectural patterns for microservices: A systematic mapping study. In: **8th International Conference on Cloud Computing and Services Science, CLOSER**. [S.l.: s.n.], 2018.

YANAGA, E. **Migrating to Microservice Databases**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2017. ISBN 9781492048824.