BEST PRACTICES FOR MONOLITHIC TO MICROSERVICE TRANSFORMATION

Buddhika Anjalee Ratnayake

158245N

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

January 2019

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date:

The above candidate has carried out research for the Masters thesis under my supervision.

Signature of the supervisor: Date:

ACKNOWLEDGEMENTS

I am deeply grateful to my supervisor Dr. Indika Perera from the University of Moratuwa Department of Computer Science and Engineering, whose help, motivation, suggestions and encouragement helped me in all the time to complete this work.

Also, I would like to thank my family and friends for their understanding during the period of the project and gave me support. Without their help I would face many difficulties in the time of the project. And finally, my grateful thanks to all the people who help me during the project.

ABSTRACT

Monolithic software systems could become a challenge in certain scenarios when the software system needs to be enhanced. A single change in the system requires full redeployment. This result in high cost of adding new functionalities while reducing the competitive advantage in the market for adding new features, since there has to be a full functional test of the application and this will increase the time a new functionality is introduced to the market. Microservices comes in to play in order to provide a solution. The study is carried out from this research aims to find the best practices when transforming from a monolithic to microservice architecture.

This research tries to find the software architecture issues and challenges when changing the software architecture of a monolithic application in to microservice architecture. Microservice architectural style is about an approach which creates a single business application as a set of simple service units. Each service corresponds to a single business process which could run independently. The communication is through light weight mechanisms such as HTTP resource APIs. One single microservice is independently deployable and build around a specific business functionality.

The microservice API is the entry point which provides the interface for multi-channel client requests. The implementation logic is hidden behind the API interface, which is in here RESTful web service API. This API accepts and process the client calls.

The proposed best practices and methodology can be effectively used in the conversion of monolithic to microservice architecture. This research also considers the practical issues a business will have to face when doing the conversion. The end goal here is to suggest a cost effective and a time efficient technique and best practices with a minimum impact to the existing business logic to convert a monolithic style application architecture in to a microservice architecture.

TABLE OF CONTENT

DECLARATION ii
ACKNOWLEDGEMENTS iii
ABSTRACT iv
TABLE OF CONTENT v
LIST OF FIGURES vii
CHAPTER 1 1
INTRODUCTION 1
1.1 Background 2
1.1.1 Issues with Monolithic architecture 3
1.1.2 Microservices 4
1.2 Research Problem Statement. 6
1.3 Research Objectives 8
1.4 Overview of the document 9
CHAPTER 2 11
LITERATURE REVIEW 11
2.1 Defining services for microservice approach 12
2.1.1 Services identification and service planning 12
2.1.2 Conceptual Elements of SOA Migration Framework 19
2.2 Web Services 24
2.2.1 RESTful web services 25
2.3 Microservices 29
2.3.1 Architecting Microservices 32
2.3.2 Disadvantages of using microservices. 32
2.4 OData – Open Data Protocol 33
2.4.1 Case studies with OData approach 34
CHAPTER 3 38
METHODOLOGY 38
3.1 Proposed Solution 39
3.1.1 Factors to be considered during the Transition 39

3.1.2 Organizational impacts of the transition 42
3.1.3 Techniques used..43
3.1.4 Refactoring process - Refactor functionality..45
3.1.5 Refactoring process - Refactor Database 46
3.1.6 Methodology conclusion 50
3.2 Evaluation Plan 51
CHAPTER 4 52
IMPLEMENTATION 52
4.1 Identifying the candidate services from the monolithic system 54
4.1.1 Proof of concept...57
4.2 Protocols used for microservices..58
4.2.1 Service Metadata Document 61
4.2.3 Handling various operations through the services..63
CHAPTER 5 77
EVALUATION AND CONCLUSION 77
5.1 Evaluation..78
5.1.1 User survey to evaluate microservices..80
5.2 Conclusion...84
References 87

LIST OF FIGURES

Figure 1 Traditional vs IT as a service [1] 3
Figure 2 Rapid growth of cloud computing 6
Figure 3 Service oriented Migration and Reuse Technique Approach[8] 13
Figure 4 SOA bridge design to dynamically expose MVC functionality through web services[9] 15
Figure 5Bridge algorithm[9]..16
Figure 6 Gradual change in generations of learning management systems. 17
Figure 7 Overview of SOA migration framework 20
Figure 8 Monolithic vs microservice architecture 30
Figure 9 Before and after using oData services[33] 36
Figure 10 Various types of testing for the monolith 40
Figure 11 Anti-corruption layer between microservices and monilithic system 49
Figure 12 Underlying functions of the anti-coruption layer..50
Figure 13 Extract services from the system [47] 54
Figure 14 A single microservice 59
Figure 15 OData service details from .svc 60
Figure 16 OData service request..61
Figure 17 OData metadata document..62
Figure 18 Functions exposed through services..64
Figure 19 OData GET request with Query parameters 64
Figure 20 Creating a new resource through OData request 66
Figure 21 Transition process 67
Figure 22 The data viewed through the microservice 71
Figure 23 Basic service oriented architecture derived from the research 72
Figure 24 CsdlAbstractEdmProvider class [37] 73
Figure 25 Metadata document for Product resource 76
Figure 26 Experience levels of the participants..81
Figure 27 Experiance with SOA or microservices 82
Figure 28 Benefits of Microservice usage with the order of importance (%) 82
Figure 29 The importance of challenges with regard to microservices 83

LIST OF TABLES

Table 1Evaluation Criteria for microservice vc Monolith 81

CHAPTER 1

INTRODUCTION

This chapter gives an outline of the main topics of the research with a brief overview of the problem and the motivation behind the research with the research objectives and the expected outcome.

In today's world of business, with the increased competition and globalization, the businesses have rapidly changing business needs, and this has led to increased consumer IT applications. The previously implemented independent monolithic systems which are designed for specific functions need to be introduced with more functionality and might need to integrate with third party applications.

Service oriented architecture is a backbone which can address these complex business needs and interconnect all the needed functionality together. Microservices helps to create an easier to configure architecture for multi-channel clients in multi-platform and will lead to cost effective operational process.

1.1 Background

Software applications has been evolving at a rapid rate with the 21st century. The business functionalities are catered more and more in a service-oriented architecture. Also, by considering the cost and the efficiency, more and more companies tend to move the cloud services rather than keeping an inhouse IT infrastructure. Now many software vendors provide services as a collection of microservices through the cloud which the businesses could use as needed.

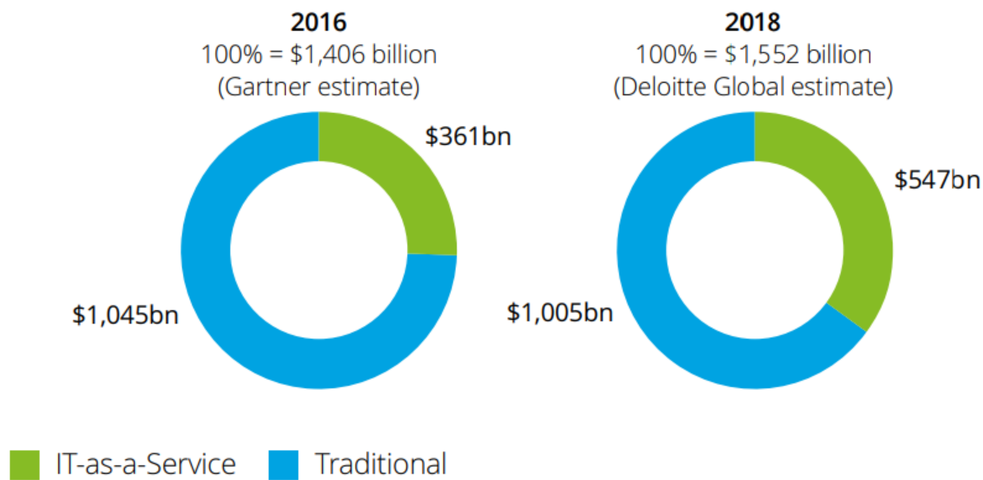


Figure 1 Traditional vs IT as a service [1]

The figure 1 presents the Deloitte global estimates [1] for IT spending market for data centers, software and IT services in billions of US dollars. As the figure shows they have depicted a rapid increase in using IT as a service from 2016 to 2018. The calculations done here are based on the Gartner estimates for total market size for IT spend on data centers, software and IT services.

1.1.1 Issues with Monolithic architecture

When a system grows in size the capabilities increase and the tightly coupled monolithic software becomes the main bottleneck in advancing the software. The highly coupled solution reduces the chance of adding more functionality and scalability. The monolithic architectural design increase coupling between the business functionalities and the module specificity decreases.

The tight coupling reduces the fault tolerance. [2]When a single module fails it could make the whole system to fail due to the high interdependency. One other issue which arises with the monolithic approach is, issues that occurs with deployment. A single change in one component would require deployment of the whole system again. This increases the cost in adding new functionalities to the existing application and is a time consuming task which would reduce the competitive advantage over adding new functionalities to the application and releasing to the market. Also when deploying the whole system it is necessary to test the functionality of the entire

application again. Therefore the cost of scaling and the time required for scaling is a critical issue in the monolithic approach.

Due to the high coupling between modules, a change done in one part of the application can create a severe bug in another module. Therefore it is not possible to go live with the new functionality without testing the whole functionality again. During this time the whole system could be unavailable to both for the business transactions and for the customers using the application. This means it also creates a monetary loss for the business organization.

When considering the organizational perspective, to handle the large amount of code in the monolithic system, it is required to assign a large team which has the knowledge on overall functionality to handle the high code base. Also the team members are highly dependent on each other as same as the modules inside the application. As an example one bug fix done by one person could have a side effect to another functionality which affect the development done by another team member. This in general implies that that development tasks could take a long time to complete.

1.1.2 Microservices

A service could be defined as a specific business activity which is repetitive and has a defined outcome. Also the internal implementation is a black box to the users of the service. The interface of the service defines how the end user can perform requests to the service. The end user does not need to know the underlying implementation details, only the response from the service does matters to the end user.

These services could communicate with each other in different ways. Microservices enable the users to create a business scenarios by linking existing services together. This allows creating coherent and decentralized systems. Also this architecture enables and eases integration of the application with different systems and to integrate new features. This indeed provides opportunity for higher scalability and reduce the cost in adding new functionalities and maintenance.

When defining microservices, it can be seen as a descendant of SOA. [3] It is an architectural style where a single application is developed as a collection of independent services which has its own process and runs a specific business functionality and communicate with others through API interfaces such as REST APIs. The microservice is bound to a business function and it is independently deployable.

Most of the business applications in the industry are developed in monolithic style that is as an individual separate system. One database consisting of many relational tables, a server-side application which connect to the database and retrieve and update data, and a client-side application which provides a user interface and interacts with the end user. When doing any changes to the system, this server-side application needs to be always modified and redeployed. The difference between this architectural style and microservices is that, these types of applications include all the functionality in a single process. But in a microservice architecture, each single unit of functionality can be exposed as a separate service. These micro services are independently deployable. So, a change in a single service only requires that specific service to be redeployed.

The individual microservices connect with each other through the APIs which are defined in according to a global standard. The outside client requests can access the microservice through the microservice interface API. This is normally a REST API. The inside implementation is hidden to the outside requests. These APIs accept calls from outside, process them and also might include some functionality such as managing traffic, handling caching and routing, filtering requests, authorization handling and authentication handling.

When defining service oriented architecture, the Cloud plays a vital role in providing on demand services for the business needed. The cloud could be basically defined as “a model for rapidly deployed, on-demand computing resources such as processors, storage, applications, and services”. [4] This provides an infrastructure on which services can be provided to the consumers. These services could be provided on different levels as required such as Software as a Service, Infrastructure as a service or Platform as a Service etc. There are many cloud providers in the industry, as Amazon Web Services, Microsoft Azure, Google Cloud Platform etc.

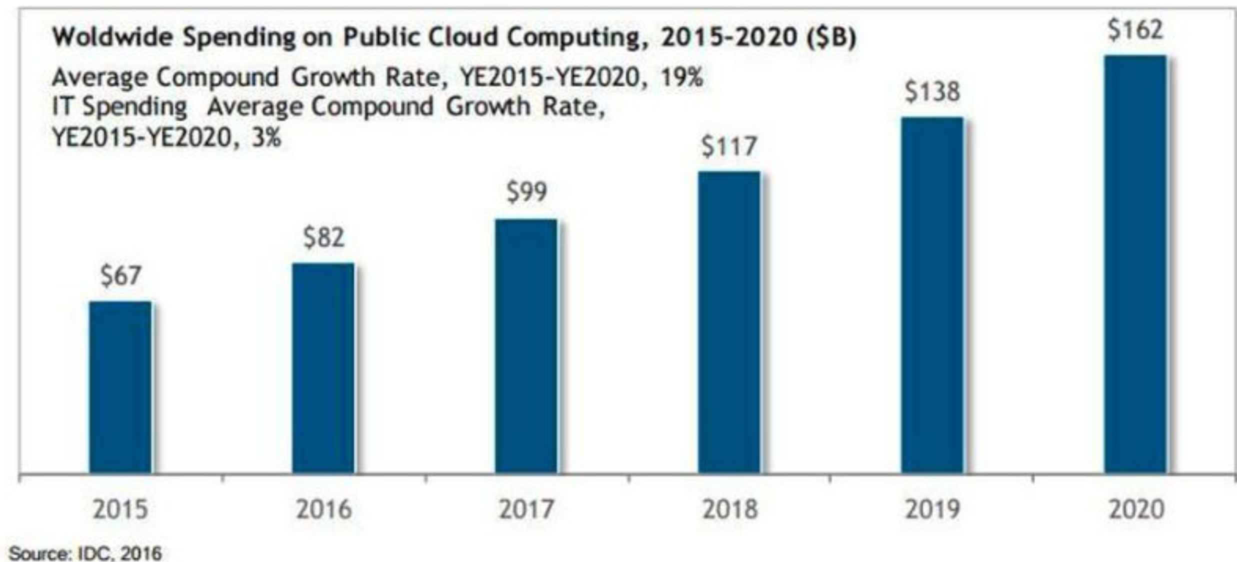


Figure 2 Rapid growth of cloud computing

[5]The impact of cloud computing and service orientation is clearly visible in the IT industry and as the figure 2 shows, by the amount of money spent on cloud computing from 2015 and predicting the number in 2020, there is and will be a rapid growth of cloud computing.

1.2 Research Problem Statement.

The monolithic architecture could become the main bottleneck in certain scenarios when advancing the software functions. A single change or an improvement to a specific functionality requires redeployment of the whole system. Therefore the cost of adding a new business functionality to the existing system is high. Also a single change done to one part of the application could introduce a severe bug in another module. Therefore a full functional test of the entire application is necessary which also impose the need for a large team with knowledge on the overall functionality.

As the monolithic approach includes many problems in the architecture, as moving forward with the business needs many other architectural patterns has been evolved. Service oriented architecture has been the most trending architecture.

The problem for the current business organizations is how to convert the existing business applications which contain all the business functionalities and currently used in the running business, but is developed using the monolithic architectures to service oriented architecture.

This will provide the businesses with more opportunities and provide them with an easy to maintain software with more integration capabilities. But the process of converting a monolithic architecture to service oriented architecture is a complex task with a high cost and time consuming. Also there is no one proper methodology in transforming a monolithic architecture to service oriented architecture. This research will look at a proper methodology for this. When trying to modernize a monolithic software with the trending technologies, mainly there are two approaches that could be followed.

1) Create the software from scratch using latest technology, or replace the monolithic system with another system or a set of systems.

2) Shift the existing system step by step to a new architectural style which support the new technologies.

Out of the two options, the second option here is the most cost effective and time efficient approach for an upgrade which preserves the existing business logic and the customizations handled specifically for the business domain. Still, this process is an expensive task which involves the risk of total failure. This upgrade could require shifting to a new software, hardware or both software and hardware.

For the approach taken from this research, that is shifting a monolithic software architecture in to a service-oriented architecture, at present there is no universally accepted model for the process. Also, no proper tools in migrating monolithic applications to microservices. It is also tough to ensure proper quality of the software application due to the lack of proper tools and techniques.

The following issues were considered during the research, [6]

1. Strategy of Modernization.

One among of redevelopment, wrapping and migration.

2. Deepness of Analysis.

The aim of the strategy is the analysis of the monolithic system for understanding its concepts and locating the important functions to be exposed as part of SOA architecture. The analysis may be superficial or deep according to the strategy used.

3. Level of Coverage.

Indicates if the proposed approach presents a complete strategy for the converting to SOA, or just a particular part of the modernization.

4. Ripeness of Validation.

Indicates if the proposed approach has been applied and validated. The proposed approach is classified as an idea, a method proved by a case study, or a commercially demonstrated technique.

1.3 Research Objectives

By considering this set of background and challenges, the expectation of this research is to create a methodology and an approach in converting from a monolithic software design to a SOA.

Even though there are many trending researches going on in this area, there are still many things to explore here. The IT industry is moving to microservice approach and it is critical for the software vendors to consider moving from monolithic standalone applications to service oriented approach.

Also, this provides a very crucial business value, since consumers in the modern world expect completely up to date services even from a smaller business.

The high-level objectives of this research is to,

- Identify set of services from the monolithic system which are independent and could be set as the micro services.

The business application which provides a complete functional solution as a whole needs to be broken down in to a set of services which will provide the independent business functionalities but as a collective will contain all the functionalities of the existing application. The loosely coupled functional areas needs to be identified according to the business domain.

- Wrapping the services into RESTful APIs with minimum changes to the underlying business logic.

It is crucial to preserve the underlying business logic of the application, therefore one objective of this research is to introduce services with minimum changes to the underlying logic.

- A comprehensive guideline on the general approach on how a monolithic system architecture could be shifted to microservice approach.

The main research objective is to provide a general approach on how the microservice architecture could be introduced to monolithic applications.

1.4 Overview of the document

This research document is divided into six main chapters, with Introduction to the research as the first chapter. Literature review of the existing work as the second chapter, Methodologies used in this project described in the third chapter, a basic Implementation as a proof of concept as the fourth chapter and Conclusion with the Evaluation and future work as the sixth chapter.

The first chapter, Introduction will first describe the background of the research study with brief details on the main concepts used in the research such as monolithic system architecture approach and microservice architecture. Then the problem statement is clearly defined with the research objectives to solve the problem defined.

Literature review of the related literature in this research area is added in the second chapter. The details on the specific areas were studied from the existing literature and these helped in defining the best methodologies and approaches in solving the research problem.

The identified optimum methodology is described in the third chapter with the mechanism for the actual implementation of the proof of concept of the research. The basic architecture used to solve the research problem is defined in the chapter defining the methodology.

Fourth chapter describes how the microservices are implemented through this research and the actual procedure of the implementation. This chapter provides in depth details on the proof of concept implementation.

The conclusion of this research is given in the chapter five with the evaluation of the results from the research study and the conclusion taken from this research.

CHAPTER 2

LITERATURE REVIEW

There are related literature carried out in this area with regards to moving monolithic applications to microservice architecture. In this section, these research ideas will be discussed and evaluated.

2.1 Defining services for microservice approach

2.1.1 Services identification and service planning

When migrating from a monolithic application to microservice approach, the initial phase of requirement gathering, and initial analysis is a crucial part of the whole process. The business functionalities which could be fine grained as specific services in the microservice architecture needs to be identified. One common problem and the main task could be identifying these cohesive standalone services from the monolithic application. This indeed will require a manual effort from the user perspective and deep knowledge of the business domain.

[7] The service planning needs to be coupled with the requirements of the business domain. This in fact needs to be considered by considering the future business improvement goals and objectives. Also it has to factor the flexibility for future changes.

One research in this area has implemented a process named SMART (Service oriented Migration and Reuse Technique) approach. [8]

Early decisions on whether it is possible for the organizations to consider reusing monolithic modules in the SOA, can be derived from the SMART process. The modifications that might be required to the monolithic components and the communications required by the SOA environment that is planned to implement is considered by the SMART process. In order to do this monolithic application is analyzed by the SMART process, the targeting SOA environment is evaluated, the service units to be developed is considered, the techniques that could be used to migrate is looked at and a rough estimate of the cost for the process and the risk that are associated in the transformation in given.

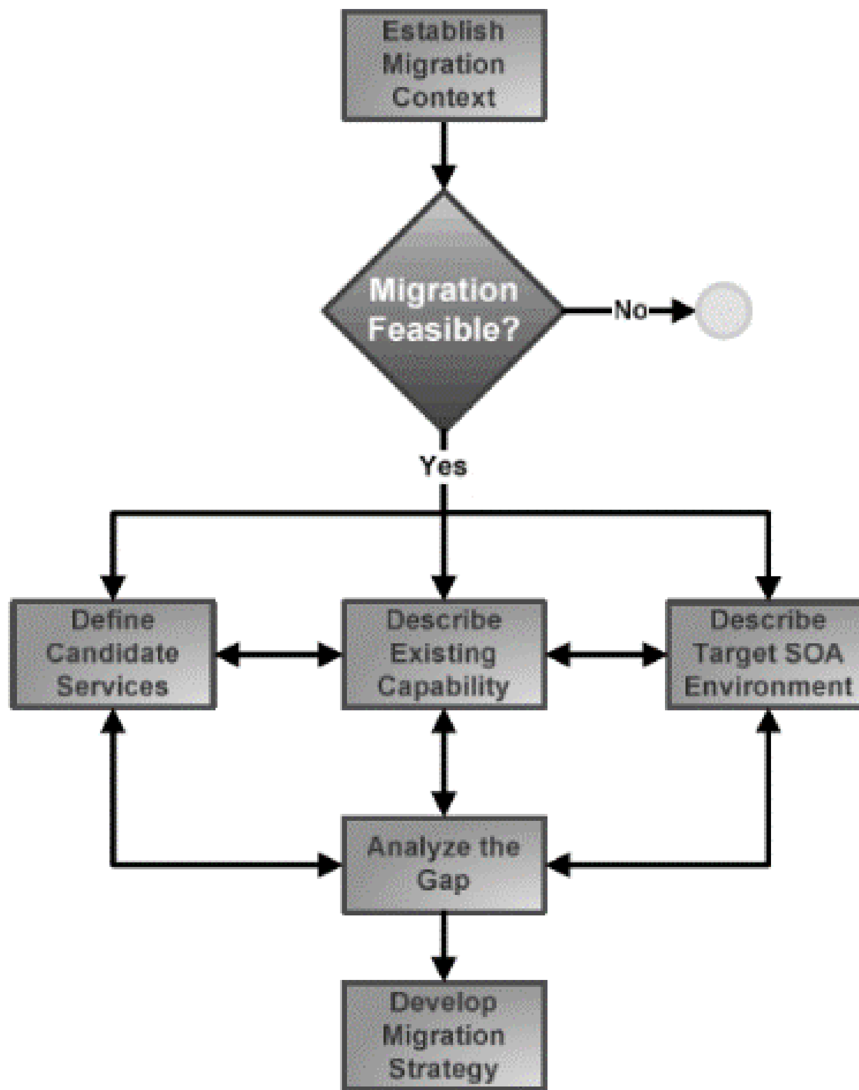


Figure 3 Service oriented Migration and Reuse Technique Approach[8]

As shown in figure 3, in here the research have mainly defined five activities,

1) Establish stakeholder context

In here the stakeholders are mainly the owners and end users of the current monolithic system. They are consulted to get a better view on the services expected from the system.

2) Describe existing capabilities

This involves obtaining a detailed data on the components of the current monolithic system. In addition, the quality of the existing system is also accessed and the initial risks that could be identified will also be listed down.

3) Describe the SOA state.

The list of potential services will be derived in here, but this may expand during the coming stages.

4) Analyze the Gap

In here, the gap between the current and the future state will be compared, and a cost-effective analysis will be carried out. Also, the risk associated will be compared with the cost calculated.

5) Develop migration strategy

Then the strategy is derived using a component service table which maps the available components to the needed services.

The basic goal of this approach is to identify a model project which could help to define a transformation plan for the organization. This will be conducted in parallel with the cost and the risk analysis.

When considering an architectural style as Model View Controller (MVC) implementation, this is a popular approach on web applications. Sabelo Yalezo [9] has suggested a technique on how functionality of a system developed through MVC style could be dynamically expose over SOA services. The MVC style implementation benefits include reusing of the code base and separating the data, business logic and presentation through 3 layers of Model, Controller and View. Here the functionality is encapsulated in the business logic layer of the system.

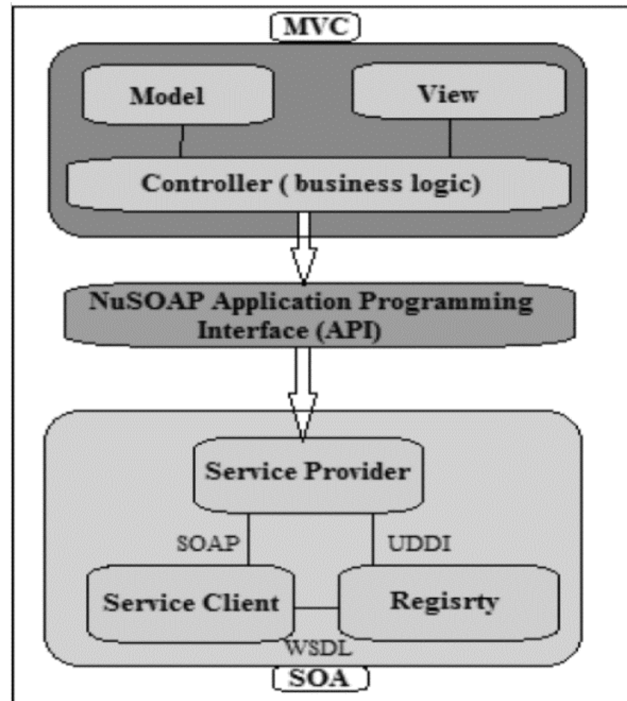


Figure 4 SOA bridge design to dynamically expose MVC functionality through web services[9]

The figure 4 defines the SOA bridge strategy which is used for dynamically exposing MVC functionality over web services. Here the bridge is connected to the controller component of the MVC platform where the business logic resides and application functionality is encapsulated. This bridge provides an interface for the service description and to consume the underlying services. When a request is received for service description the bridge will automatically go through the available controllers, then find the functionalities that are provided by the controller and get details on what are the data types and message information that is needed for the WSDL service description. This information is given as a WSDL response. The same way all the requests are first handled by the bridge and is converted into requests for the MVC architecture platform.

1. *Get all the available controllers in a MVC application*
2. *For each controller get the available actions*
3. *For each action get the arguments*
4. *Type = argument data structure*
5. *Messages = action + controller name*
6. *Check message types:*
7. *if one way message : input and message output*
8. *if request-response message: input only*
9. *if notification message :output only*
10. *if solicit-response message :output then input*
11. *PortTypes = controller name + action*
12. *Binding = protocol supported by the bridge*
13. *Service = bridge location and URL*

Figure 5Bridge algorithm[9]

The figure 5 shows the bridge algorithm used in the research, and even though this algorithm shown here seems specific to the MVC platform that was used in this research, this could be adapted for other MVC architecture applications.

Semith. Cetin, N and Ilker Altintas [10] speaks about MigrAtion to Service Harmonization compUting Platform (MASHUP). A key idea in here is to achieve integration even at the presentation layer, not only at backend layers like application or data. In here there are mainly six migration strategies to address both the behavioral and architectural aspects of the migration.

Step one is to model the target enterprise business requirements. The business requirements are modeled for understanding the functional requirements of the target system. Step two of the process in Analyze. In here, the existing monolithic system is analyzed to identify the domain specific needs and to investigate the reuse of potential existing components. The outcome from this stage will be details on the architecture of the system, components of the system, details of the infrastructure, characteristic details with presentation, details about quality, interfaces, maintainability level, complexity analysis, coupling and etc. Step three includes mapping the functional requirements to the software components and define the services for each component. These components could be from the previous monolithic system or implanted from scratch. In this iterative process, if a component already matches the business requirement, then it could be

simply wrapped up. But if there is a gap between the component and the requirement, then the gap will be filled during the service wrapping. If the gap cannot be filled by this, then a new component needs to be developed for those requirements. This could be further broken down into sub services. Step four is designing a "Mashup Server Architecture" with domain specific kits. At the end of this step, the service mapping from the existing components is done and these are prepared to execute through "Mashup Server Architecture". Now, in step five, Service Level Agreements will be defined. Then in the last step, that is step six is implementing and deploying the services. This consists of wrapping of existing components, customizing existing components with additional requirements and developing new services. This process is mainly implemented to overcome problems related to Quality of Service.

Service Oriented Architecture define a concept which describe the business process and logic as individual services and publish and expose the functionality in a standardized way which makes it possible for the other services to integrate with these services in a flexible manner.

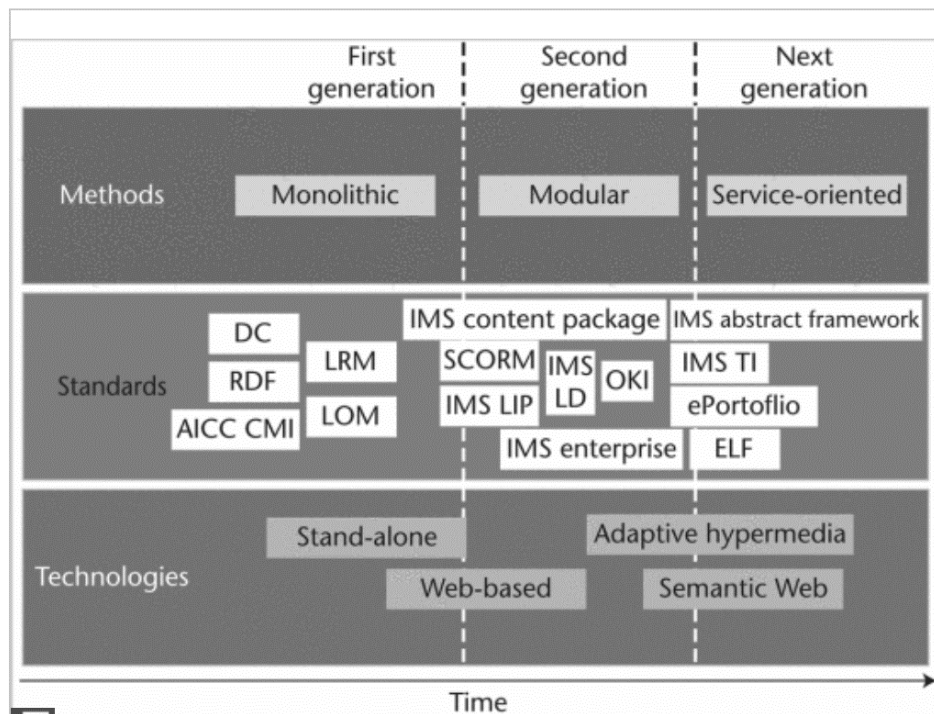


Figure 6 Gradual change in generations of learning management systems.

[11] The service oriented e-learning platforms has evolved to be more SOA and as figure 6 illustrates there is a gradual change in generations of learning management systems. The first generation support only the content only inter operation. The user's perspective and their user profiles were taken into account in the second generation and the next generation support targeted personalization.

When moving one step down from the monolithic approach we could target at the component oriented architecture, but this is again more tightly coupled that service oriented architecture. Even though this could be a starting point to SOA, in here the changes to individual components will have an impact on the components which has a dependency with the modified component. This makes the component oriented architecture less flexible than the service oriented architecture.

One of the first approaches that was introduced to systematically investigate the criteria to be used when breaking down a large system into smaller modules was focusing on doing so with information hiding technique. [12]

[13]The authors define a systematic approach which requires manual tasks to identify the microservices in a monolithic enterprise system. They construct a dependency graph between the server and the client modules, database tables and business functions. One other attempted methodology is ServiceClutter [14] This is a concept of coupling cards with an extensible service decomposition toll framework architecture that integrate graph clustering algorithms and features priority scoring starting from nanoentities and nine types of analysis and design specification(including domain models and user cases). But one drawback that can be seen with this approach is it is not possible to extract or construct the needed structure information directly from the monolithic application itself and therefore need to rely on the user to provide the software artifacts in the defined expected model.

Migration of monolithic to SOA embodies a key challenge of service engineering, the rehabilitation of pre-existing enterprise assets into a service-based system. This paper proposes a conceptual framework to illustrate such a migration process based on the definition of SOA migration. This framework facilitates the representation of the SOA migration processes in a unified manner and therefore provides the basis for their comparison and analysis.

The SOA migration framework addresses the question of "what does the migration of monolithic systems to SOA entail". In the context of architectural recovery, a conceptual "horseshoe" model has been developed by the Software Engineering Institute, which distinguishes different levels of reengineering analysis and provides a foundation for transformations at each level, especially for transformations to the architectural level. Given that migration is considered as a reengineering process and that the horseshoe model is a generally accepted conceptual model for reengineering, this paper propose an extended form of the horseshoe model as a holistic model of the migration process.

This horseshoe model recovers the lost abstractions and elicit the fragments that are suitable for migration to SOA, alter and reshape the abstractions to service based abstractions and finally renovate the target system based on transformed abstractions as well as new requirements.

2.1.2 Conceptual Elements of SOA Migration Framework

The migration process could be defined as the set of tasks carried out during migration.

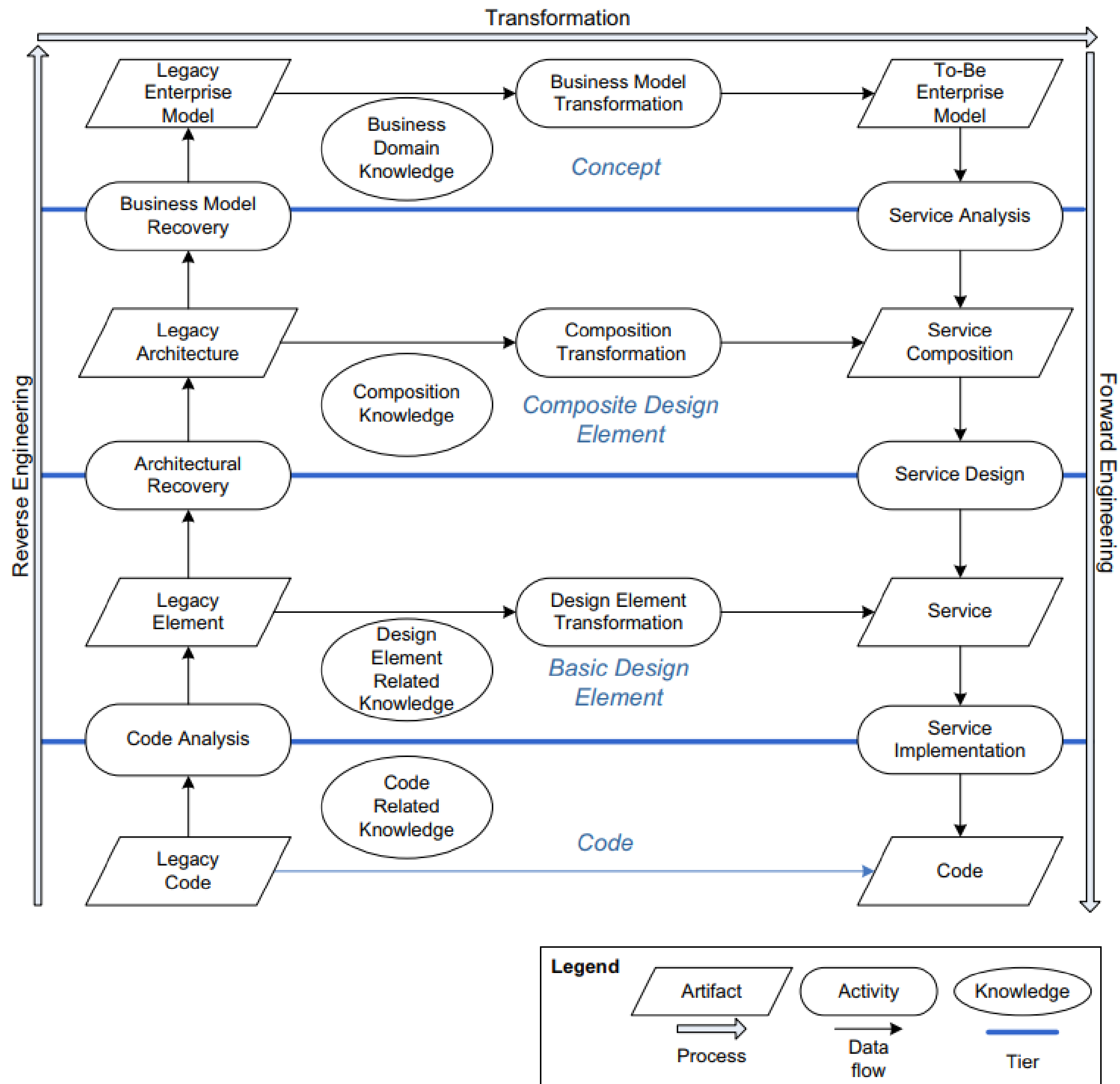


Figure 7 Overview of SOA migration framework

The migration process shown in figure 7 generally consist of three sub-processes, [15]

1. Reverse engineering

The procedure of evaluating the current system to recognize the system's structure, processes and actions and to build an illustration of the system in different form or at an advanced genre of abstraction. When considered from the transformation angle, the key objective of the reverse

engineering process is to get an idea of the monolithic architecture to the degree to identify the finest candidates for services in the current monolithic application elements for transformation to SOA.

From the overall handling viewpoint, this procedure could begin from current development and carry on with getting the design entities, fixing the architecture and trying to capture abstractions in requirements or business models again.

2. Migration

This is the method of transition from one architecture form to another while keeping the same level of abstraction. From the transition angle to SOA, the transition procedure carry out transition of monolithic modules to service based units and is done through a set of tasks linked to different levels of abstraction.

3. Forward Engineering

The resulting structure is re-implemented with considering the current needs and objectives given by the transforming service bounded architecture as well as the target objects createdd during the transition process.

- Artifact

  Artifacts are the resulting objects created from the process. These can be a product output from reverse or forward engineering and transformation process.

- Activity

  An activity is single production step of a methodology which tries to create or change a specified set of artifacts. Activities shows the stages on what should be done in each stages of reverse engineering, transformation and forward engineering processes.

- Knowledge

  Knowledge can be described as the term defining complete set for concepts related to transformation method: procedures, principles, data, techniques, models and context. Based on the knowledge type, different approaches can be defined.

SOA transformation platform is which presented by Razavian [16] enables to describe and isolate the characteristics of migration plan in terms of methods it supports, artifacts included, activities carried out, and types of knowledge exploited.

A details evaluation of the present literature in the category of monolithic system transformation techniques to SOA is presented by Almonaies in 4 categories. [17]

1. Replacement

Alter the current application or substitute it fully with a software available in the market. Replacement can occur either as in one go or in an incremental approach. If the monolithic system is very stable and has a good quality structure it is better to go ahead with the incremental approach. There are also risk involve with replacement since the new system is not familiar and all the functionality that were supported in the monolithic system might not be supported in here. Also the maintenance can also be an issue due to unfamiliarity.

2. Redevelopment or Reengineering

Redevelopment and Reengineering technique can be used to slowly present SOA design to monolithic systems. This needs studying the existing system and doing alterations to convert it to SOA. This will require activities such as redesigning, restructuring, reverse engineering or re developing new systems.
There are also issues such as identifying services from the existing system, since this is not a simple job.

3. Wrapping :

Wrapping is essential to construct a way to expose the services in SOA so that these will be visible to the other services from outside. This can be a good workaround to develop services while the monolithic system remains intact. If the monolithic system has good quality code and the high important business functionalities then wrapping can be a good

choice to provide API interfaces to the monolithic system. But this will not change the underlying implementations of the monolithic system and therefore it will not fix any existing issues of the monolithic application.

4. Migration :

Migration moves the monolithic system to the more flexible SOA environment while preserving the original system's data and functionality. In migration strategies code segments identified, decoupled, and extracted using approaches similar to those used in wrapping and redevelopment. User interfaces are then reengineered to be compatible with an SOA structure. Migration strategies incorporate both redevelopment and wrapping and aim to produce a system with an improved SOA compatible design. It is not always obvious how to distinguish migration approaches from wrapping and redevelopment techniques.

Above mentioned approaches are compared using eight criteria,

1. Modernization Strategy
2. Legacy System Type
3. Degree of Complexity
4. Analysis Depth
5. Process Adaptability
6. Tool Support
7. Degree of Coverage
8. Validation Maturity

Even though there are many advantages in migrating from legacy applications to SOA, it might not be beneficial if it was done in a way which could introduce bugs to the system. Therefore the transformation technique is quite important. But it may be such that there is no one clear technique which could be applied to all the situations. The approach for any application to be transformed needs to be looked at from the individual system perspective by comparing the strengths and weaknesses, whether the approach is feasible to be applied etc.

The highest risky task in remodeling a monolithic application is selecting the services to be exposed in current code base, and there is a requirement for improved tools and techniques to find functional value in huge code bases. There is a necessity for objective metrics to compare the methodologies, and most methods does not consider the service implemented in the importance of quality characteristics, such as performance, security and reliability.

With these finding it is clear that there has been some prior work in this area trying to extract the fine grained services from a monolithic architecture. This involved software metrics, repository mining or static analysis.

2.2 Web Services

An application software can used via the network and the application's data is exchanged via URL queries in a web service implementation.

The two main web services in the industry are RESTful web services and SOAP based services. When these two are compared, as a general consideration it could be seen that RESTful services are evidently becoming more popular with the time and in the trend of replacing SOAP based web services in many fields. This could be related to the fact of ease of use and deployment and light weightiness of RESTful web services [18]

The work [19] presented by Gavin Mulligan compares the SOAP and REST architectures for an interaction independent middleware framework. After the implementation and evaluation of both the RESTful and SOAP architectures, it has come to a conclusion that REST architecture style is also more effective for the bandwidth utilization, services access and request latency. It also mentions that REST implementation is more effective for CRUD operations over SOAP.

[20]Another research of comparison of SOAP and REST web services provision technique for mobile applications similarly proves and present results that services built on REST to be much

better approach for mobile application, but when it comes to the security, this has mentioned that SOAP will provide communication which is highly secure than the REST protocol.

2.2.1 RESTful web services

For applications which needs to create API to expose the underlying business functionalities to the outside as services, REST may be a better approach since it is easy to maintain, light weight and it is much easier to scale the functionality on top of REST.

[21]In the RESTful web service construction, resources can be exposed through referencing the resource name in the URI through the HTTP protocol and by using a HTTP method, GET, POST, PUT and DELETE to handle the resource on the web server. Resources are references through uniqur URLs. for example, "http://hostname/service/group/id=2002." Also since the resources are exposed through the URI, the RESTful web services supports server-side operations, caching, parallelization and load balancing.

[22]A study carried out for reengineering monolithic systems with RESTful Web Service identifies following features as the main participants in a service oriented application.

- Service Provider
  This expose the services with well-defined information on the resources and is the main owner of the information exposed.
- Service Integrator
  This will provide a suitable service framework to the users accessing the services using the underlying software systems platforms. It can derive the services based on the information on the resource metadata.
- Service Users
  The final users who used these exposed services and the application which uses the services will be the service users.

The three members defined earlier in a service architecture can modernize their current application with business services to use the current functionalities and create new functionalities with very little implementation struggles. Feasible and beneficial modernizing practices and platforms are going to be more and more vital, which can expose the current features to give services with no extra framework or technical limitations.

These could be defined as some main decisions on moving to serviced oriented architecture.

- Functionality reuse
  Without implementing services from scratch which will need more effort, the existing functionality could be wrapped to expose them as services.

- Service Capabilities mining
  The service user's functional needs will be the main drive for implementing the business as service with combining the skill of organizational technical and service proving abilities.

- Future mash-up
  With the introduction of Web 2.0, mash-ups present a capable fresh implementing methodology to create combination of systems and facilitate easier, quicker combination of current web based systems.

- System Maintain and update
  The practices for modernizing existing monolithic applications and wrapping approaches can also be since this makes it easier to maintain and update the system.

Few of the main problems in modernizing existing monolithic applications with RESTful web services can be listed as,

1. Finding and discovering resources correctly.
2. Build URIs which are scalable and platform independent.
3. Defining functions and actions.
4. Be mindful on the changes that needs to occur in near future.

5. Metadata about resources.

To overcome the problems mentioned above, it is possible to use these techniques

1. Find potential Web Services.
2. Create composite web services built on current interactions.
3. Define services and explain usual procedures.
4. Plan demonstrations and create RESTful web services.
5. Wrap current functionalities with RESTful web services.

If a data driven system or an entity driven system is considered to be modernized through SOA there are problems that needs to be considered like, dependency between the presenting the resources in the URLs. A metadata type of entity can be derived to include the information on the other RESTful resource web services.

Several classes which can be used to categorize RESTful web services. [23]

Type I: Resource Set Service

The web services linked and derived from business functionalities can be defined in this class. "In a preventive maintenance application on objects, resource related to a set of maintenance orders and serial objects is able to define as of this type. We can define them as MaintOrderSet and SeriaObjectSet respectively." The HTTP GET, PUT, DELETE and POST requests are supported by the web service.

Type II: Individual Resource Service

Resources of a particular area are derived in this type of service. An individual maintenance order and individual serial object is possible to be mapped this category." This category will provide three HTTP requests that is GET,PUT and DELETE

Type III: Transitional Service

This category is to consume the service resources, generate new resources or modify the current resource or to change the state of the existing resource.

Three fundamentals for configurable RESTful service is described by Shang-Pin Ma and Chun-Ying Huang [24]. That is service process design, data composition, and widget presentation design

1. Service Process Design:

The first phase is to plan the order in which a series of RESTful services will be implemented and how to interrelate among these services. Two things are required in this method. Main document is, a RESTful Service Description Document (SDD), to state the purpose of the service and next a Mashup Document (MD) to describe the sequence in which data communications happen over various services.

2. Data Composition:

It is vital to fulfill the end user requirements of information over the data exposed through services. Data composition method is created to fulfill this condition. The fundamental idea on data composition was copied from the join operator in the area of databases. After data composition, the exposed service data sets is possible to be organized using Merge Scan and Nested Loop, as a result of which several amount of organized/un organized data collections can be arranged for presentation in Widgets.

3. Widget Presentation Design:

Throughout the Service procedure design stage and the data composition stage, the key factor was on service introduction and data management. Nevertheless at the end, the service accessing user needs to get the data set in a format which is easily readable and understandable. XML or JSON representation of data is not a good format for this. For this "Widget Mashups is used, which chooses data sets to be presented and defines which widgets are essential to expose the choosed set of data.

2.3 Microservices

Microservice is an architectural style which views a full application as a collection of services which are,

1) Independent services providing a specific business functionality.
2) Loosely coupled services.
3) Easy to maintain and test each individual service.
4) Possible to deploy each service independently.

Microservices introduce many benefits to the architecture since the services are independently deployable which provides more scaling options. The big organizations in software industry like Google, eBay and Netflix and many other businesses have implemented the microservices architecture moving forward from the monolithic architectures. [25] [26] [27]

There is no unique definition as such for the term microservice. But we can define a few characteristics which is a fundamental in the design of microservices.

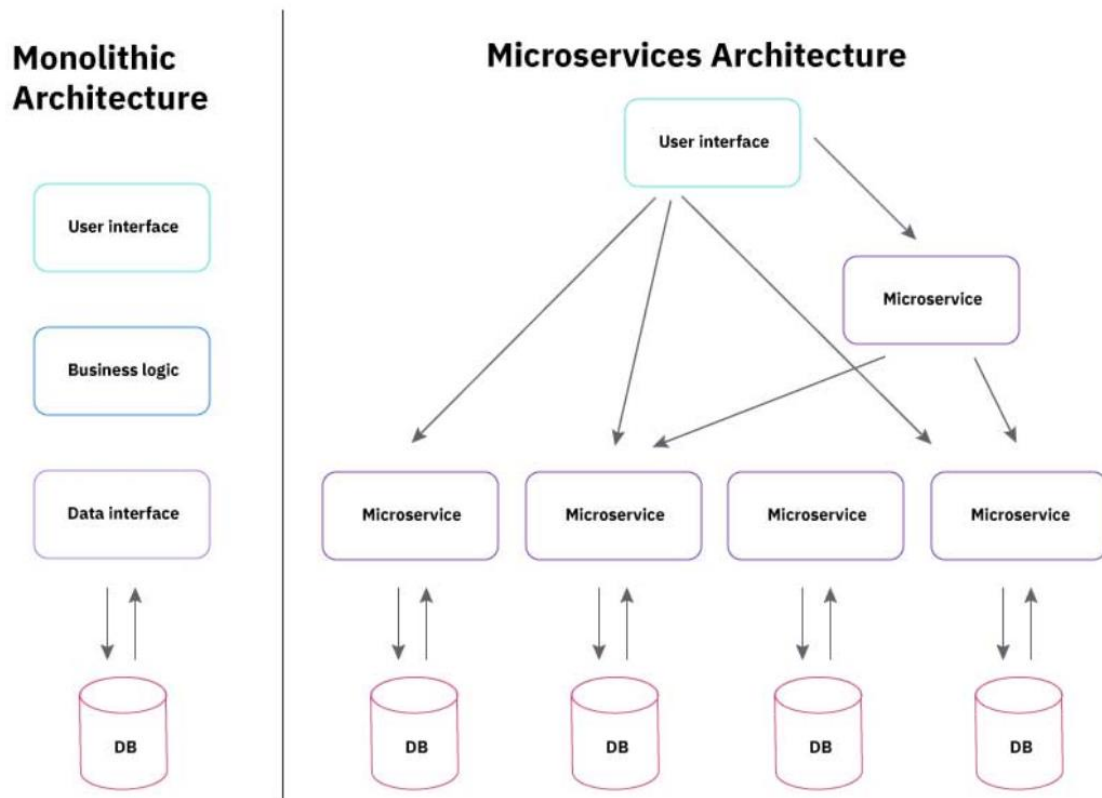


Figure 8 Monolithic vs microservice architecture

[28] Figure 8 shows in a diagrammatic representation of the difference between the monolithic architecture and the microservices architecture. These microservice specific characteristics could be listed down as below. [29]

1) Multiple independent modules

Each individual microservice is possible to be deployed independently and after a modification or an improvement done to a single microservice, it could be redeployed independently without disturbing the integrity of the total system. Because of this reason it is not required to redeploy the full system again after an alteration is done to a particular service. But this method could also have high complications when the functionalities are redistributed between components.

2) Each microservice corresponds to a specific business function.

In microservice implementation methodology one or many microservices which relates to business functions are created and maintained by each and every team. The corresponding final service is owned by the particular team.

3) Simple Routing

Microservices create a response according to the business functionality when they receives requests through different sources and process them. A specific business flow is processed through each microservice and it accepts the input, process the data and by applying the business functions returns the business output.

4) Decentralized

A unique service manage a separate data base in the microservice architecture presenting a decentralized nature, as opposed to the monolithic applications which has a one database storage that keeps the data for the whole application.

5) Fault resistant

It is expected from the microservice architecture style that even a single service fails, other services to communicate efficiently and function even though a specific function has failed in the system. Due to this need microservice architecture could be more complex when compared to the monolithic architecture.

6) Evolutionary

When considering the future trends and technologies microservices are the best approach since it will be able to integrate with the evolving technologies which is another positive aspect when compared to monolithic architecture.

[30]J. Wang and F. Fahmi introduces an approach where the application system could be decomposed in to several subsystems recursively and then define the microservices accordingly. The MOOS approach which is introduced here is a systematic approach to developing software with layered microservice using object oriented specifications. In this approach the system is

decomposed into subsystems repeatedly and then microservices are defined for the subsystems accordingly. With the analysis that had been carried out during the research it shows that MOOS increases the concurrency of the application while improving the performance and increase the reusability of micro services.

2.3.1 Architecting Microservices

The next era of Service Oriented architecture could be defined as Microservices architecture. But when we compare service oriented architecture and micro services architecture some differences also can be found in these two [31]

- Service oriented architecture implements the viewpoint that it is best to share the resources as much as possible in order to encourage reuse of resources. The strategy of services in and microservices architecture is focused by a share nothing viewpoint so as to provision agile approaches and encourage separation and independence.

- Microservices architecture primarily concentrate on service composition, while service oriented architecture depend on equally on service arrangement and service composition.

[32] Microservice architectures are mainly appropriate for cloud structures, as they significantly value from the elasticity and fast accessing of business services. Cloud structures and new technical advancement perform an essential part for building microservice platforms and maintaining the related issues and difficulties.

2.3.2 Disadvantages of using microservices.

On one hand microservices provide much better options and is a more effective approach, but by implementing microservice architecture, the communication issues, cross product team development issues and any other issues which were hidden in the earlier monolithic architecture

approach can now be forced out in the open. But this again in one hand can reduce the development time, testing time and the time to release.

But in general whether the microservice architecture is the most appropriate approach for the specific business scenario depends on the specific requirements of the business.

There can be several disadvantages as well of the microservices depending on the scenario where it is used.

1) Since the deployment is distributed the testing and quality assurance tasks can be complex to perform.
2) Increasing the number of services will limit the information accessible through a single service implementation.
3) It could add an overhead on the development, since the developers need to consider the different message formats of the services, load balancing and handling fault tolerance.
4) Also the nature of the services could also introduce some duplication effort.
5) When one whole application is broken down into services, if the number of individual services increase, integration of these services and managing the product as a whole can be complicated.
6) An additional effort must be given to implement the integration between individual microservices.
7) If the development use case is cross product development, which requires several integrated services to be modified, it could be necessary for multiple development teams to communicate and collaborate together to complete the development task.

2.4 OData – Open Data Protocol

A standard but a simple mechanism of creating and using RESTful APIs is derived from open protocol named OData protocol.

- The [OData-URL] -To find the data and metadata visible by an OData service as well as a collection of marked URL query capabilities, this defines a set of rules for constructing URLs.

- The [OData-CSDLJSON] – The data model of the entity as visible through the OData service is defined in a JSON format through this description.
- The [OData-CSDLXML] - The data model of the entity as visible through the OData service is defined in a XML format through this description.
- The [OData-JSON] – The resources that interact and communicate through OData is presented in the JSON format through this description.

Through OData protocol the metadata information about the data models can be viewed along with viewing data and modifying data through OData. This also provides,

- Metadata: The description of the resource exposed in minute details could be found in Meta data document.
- Data: The collection of entities of data and the associations among those.
- Querying: Viewing the data after requesting for the data which querying for a specific function.
- Editing: CUD operations, Create, Update and Remove data.
- Operations: calling to the business functions
- Vocabularies: A list of derived terms.

Simple and Uniform interfaces are required for REST architecture. Since OData is based on the REST architecture, the resources of can be accessed from clients through unified interfaces.

2.4.1 Case studies with OData approach

SAP gateway

SAP gateway present SAP Business Suite features as REST-based OData services. It permits SAP requests to expose data through extensive variety of devices, technologies, and frameworks in a technique that is simple to realize and consume [33]

Reasons for using OData is,

1. OData is created on standards and it provides access similar to the access through database and built on REST architecture.
2. If OData is used, it is not necessary for the outsiders to be knowledgeable on SAP architecture to consume the services.
3. This is flexible and can be used on top of many platforms.

There are also some advantages in using OData as REST services,

1. The web browser could be used to visualize the end output.
2. Applications used in stateless nature.
3. Interrelated data is received and one data can point to another.
4. Use simple requests such as GET, PUT, POST, DELETE, and QUERY.

SAP Gateway is in SAP application layer. It perform as a connection to the external users to link to the SAP application information. External users request through HTTP requests and SAP respond with OData over the Gateway. So the SAP application data can be requested through OData by any devices, frameworks and other applications. XML and JSON technology is used to create the OData service interface. It is possible to communicate through the web since these are simple text protocols.

The external system and the server can be using two different underlying technologies such as two separate programing languages, but still the communication is possible since it's through HTTP.

General idea of this design is server hosting the data and clients requesting the data to read and operate by calling to the server. To request the service, server needs to exposes one more or endpoints. Clients only need to know the server side endpoint in order to call the service.

Following figure 9 shows how SAP architecture has changed with the use of OData,

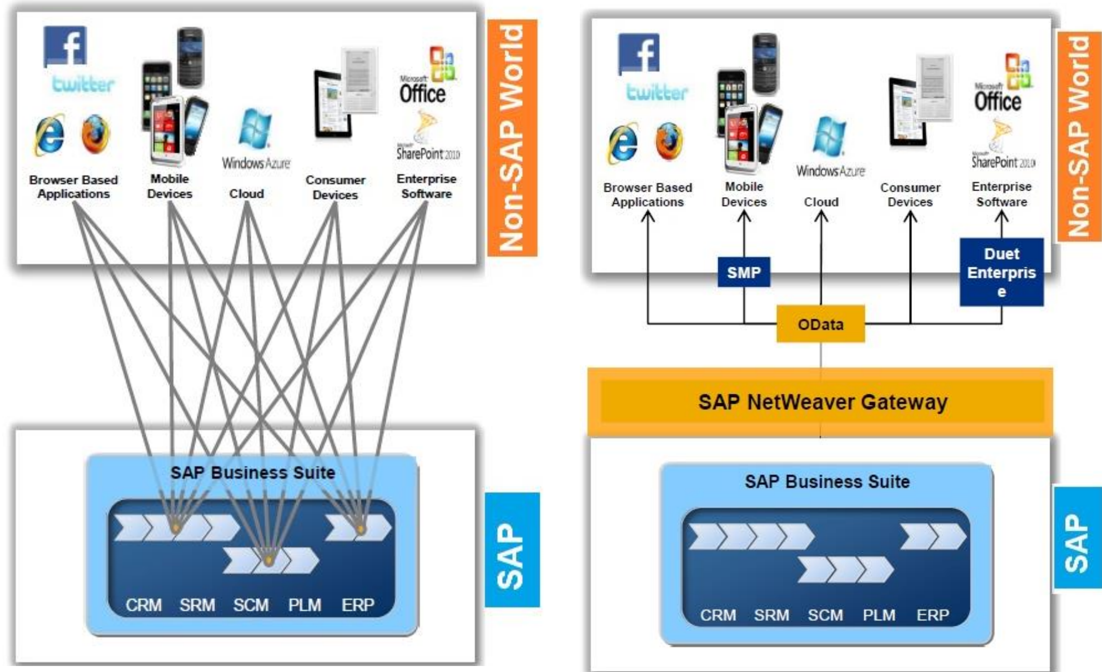


Figure 9 Before and after using oData services[33]

This figure 9 clearly shows the difference in two architectures. Before using OData architecture looks messy and it involved lot of work to integrate SAP data with the outside world. Developers needed to have the SAP knowledge in order to do the integration. With the use of OData architecture looks cleaner and it involves let effort to do the integration.

IBM Data Server Gateway for OData

IBM Data Server Gateway allows to rapidly generate OData RESTful services to view and modify data. [34] IBM Data Server Gateway is built on Apace Olimgo V4 on top of OData version 4.

IBM Data Server Gateway for OData makes HTTP data services, which let resources recognized using URLs and derived in an abstract data representation, to be distributed and modified by Web consumers by means of simple HTTP requests. This offers vibibility to data from any framework or device without needing any new client software.

Some Key features are,

- Supports all data types except LOBs and XML.

- Supports CRUD operations.
- Publish OData V4.0 REST API endpoints/Services for selected database tables.
- JSON and XML request/response payload are supported
- Persistence of REST API endpoints/services in Derby database.
- Authentication and Authorization during OData REST API endpoints/services generation.
- SSL support

Some advantages of IBM Data Server Gateway for OData,

- Using IBM Data Server Gateway can benefit from thr consistent web protocols to transfer data in and out and the user can construct and application without a database driver included.
- In the current cloud and mobile platforms DB2 standards can be applied.
- OData is interoperable and not specific to a single platform. Therefore data can accessed through any platform including mobile applications as well through REST APIs.
- During the project development cycle it is not necessary to spend time to create, document and support APIs. This will make the development more effective and efficient and the data will be exposed in open standards which results in easy accessibility.
- It is not necessary to have a middle tier, since the only transition to do is it to transform the data in to XML or JSON format. Therefore it decrease the complexity which arises with layered application architecture.

CHAPTER 3

METHODOLOGY

3.1 Proposed Solution

The main objective of this research is to introduce a general approach by which a monolithic software design could be turned to a Service Oriented Architecture through the concept of micro services. In here, a cost effective and a time efficient technique is considered which also will gives a minimum impact to the existing business logic and the database.

It is not an easy task to transform an existing application from monolithic architecture to microservice architecture. There are various factors to consider before going ahead with the transition from monolithic to microservices architecture. After these initial factors are actually fulfilled, the real process of transition can actually begin. The risk of transformation is high when the code base of the existing application is quite large. In that situation the transition should happen in small steps to reduce the risk of failure. As the first thing the code base needs to be analyzed and the correct parts needs to be picked up for the process. There are several factors that are needed to consider when selecting which business functionality to be selected for the service transition. Some of these factors are whether the existing functionality needs enhancement of current technologies, whether it includes any functionality that needs change in the immediate future and functionalities which are not inside the deep of core of the application. Since it will be beneficial to start with a functionality which already needs a functional modification, than starting with a stable static part of the application.

After the part of the code base that could be transitioned to a microservice is selected, then the splitting process can start. All the business functions and the data related to that part of the business process needs to be taken out from the monolith and needs to be applied to the microservice. The communication between the monolith and the microservice should also be taken into account.

3.1.1 Factors to be considered during the Transition

Before the actual process of converting a monolith to microservice can begin, there are several factors that needs to be looked at. First the business organization needs to be adjusted to accept this change in technology. There should be a knowledge base in the organization with the relevant

technological expertise. The developers need to be able to understand the technical implementations and any issues with these.

Also one main factor is how to test the microservices once the microservices are actually implemented. Proper testing mechanism needs to be sorted out and planned initially before the actual transition. As the first stage, it is needed to look at the test coverage of the monolithic application. If the whole application is actually tested manually, this could be a big bottleneck for the microservice.

Microservices could benefit from using automated testing. Therefore it would be better to start to have automated testing for the business process that was selected for the transition. Microservices can be released frequently if automated testing is enabled.

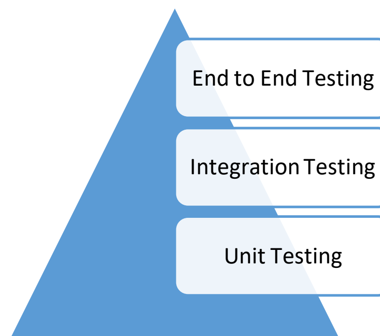


Figure 10 Various types of testing for the monolith

As shown in figure 10 for the monolith, different types of testing could be used such as unit testing, integration testing and end to end testing. These could be used to ensure that the business functionality remains unchanged once the microservices are built. Each part of the code base needs to be unit tested which gives an individual feedback to the developer and it is faster to run. Since the testing scope is smaller for the unit tests, it is much faster than the integration tests. But the integration tests will provide a bit higher scope with the business functionality and will give a better idea about the application status. For much bigger scope, end to end tests with the user interface, needs to be performed. This test will take a long time to execute but will provide an overall idea on the functionality.

If the integration tests and end to end tests are all created and can be properly executed, these will be very useful once the microservices are created to ensure that the functionality remains the same. If these tests are not properly defined during the transition process new issues could have been introduced and it is likely that these will go unnoticed initially.

The process could be much easier if the technologies used to develop the microservice is the same as the underlying technologies of the monolith. Then the unit tests of the monolith could be refactored and reused for the microservice. These tests could be used to individually test small code units of the business functionalities even after creating the microservice. These will eventually be more useful when adding new functionalities to the microservice as well. If the underlying technologies used for the monolithic and microservice differs, still the created unit tests could be used as a reference to create new unit tests for the microservice. [35] Contract tests for the new microservice could be created using the integration tests that are testing the interface of the monolithic application.

The integration tests also might need a modification, since the interface which is connected now can be a REST interface or OData interface. Still the business scenario will remain unchanged and that should be tested. When considering from the business scenario, since the functionality needs to be the same, the interfaces of the monolithic and the microservices will be quite similar. As an example the business interface which had user registration process will remain the same with the same validations when moved to the microservice as well.

The scope of the end to end testing is high and will cover a bigger code base and can give a better understanding whether the functionality works as intended. This will be more important once the microservices are created for the monolithic application. The development team can depend on these end to end tests to finally ensure the microservices function as needed. But one complication with the end to end tests is, this will require all the microservices to have implemented to enable the end to end tests to run. If needed to be integrated with the monolith, this might complicate the testing scenario.

3.1.2 Organizational impacts of the transition

Transition from monolithic application to microservices is a quite a huge task which could affect the business organization as a whole. Depending on the complexity and the size of the code base, the time taken for this transition could vary and sometimes this could even take years. [36] To achieve this the development teams will have to adapt to new technologies and have to learn new skills and techniques and also both the newly implemented microservices and the monolithic application needs to be maintained parallel during this time.

At the initial transition stage a dedicated team needs to be assigned to handle the new development of microservices and to understand the challenges that could arise and the technical limitations or challenges the new architecture involves. Then in the latter stages this team could support all the other teams with the infrastructure framework as working as the platform team. [37] Architecture to build the message queues and to derive a suitable technique for monitoring and logging and other general guidelines could be defined by this team. Since these are not specific to any one product or service, it is not necessary for each individual team to derive these on their own. This team could act as the framework team which handles the platform issues and act as the general platform team even after the whole transition is completed.

The transition process will be slow at the beginning, since the development team get adjusted to the new technologies and the new architecture. Also delivering new features at the beginning will be slow since some functionality needed to integrate with the new functionality could still exist inside the monolithic application and some refactoring to the code base might require before the new development. This could make the delivery of new functionality late to the market at the beginning.

Taken aside the technical limitations and challenges that could arise during the process, there will be organizational considerations as well that needs to be taken into account. New skills will have to be recruited and sometimes the organizational structure also might need to be changed in order to achieve this change. [38] The monolithic applications are normally limited in the technologies that are applied, but this will change when the microservice architecture in implemented and different programming languages and different underlying technologies can be used in the

microservices. Therefore these expertise might need to be hired or trained and divided among the cross product teams to strengthen the teams with the relevant knowledge.

Since the microservice architecture might require a different organizational structure. This might also need to be taken into account [39] One example could be a scenario where the teams were previously build on the combination of the different roles of the team members. This might have to change and the different skill sets of the individual team members might need to take in to account when microservice based architecture is developed. The teams should be individually capable of fully developing, testing, deploying and maintaining a microservice based functionality.

3.1.3 Techniques used

When deciding on where to start the process of transition, it would appear that the best place to start is with a known existing functionality and transform that to a microservice at first. But it will be wiser to start the process with a new functionality and create this newly through the microservice architecture. [40] According to previous literature examples, building a new functionality from the base will be easier than trying to refactor and develop from the existing monolithic application. A decision could be taken at the beginning that all the new functionality to be developed will be implemented based on the microservice architecture and no added functionality to the monolithic application unless it is a critical bug fix. This will increase the number of microservices generated with time. [41] To handle the refactoring of an application architecture, this might be the best possible approach that could be followed. From the experiences gain from this development the teams could understand the challenges of the microservice architecture in more detail. Since the core business solution is still residing inside the monolithic application, the impact of handling the microservice architecture initially will not affect the most important business functions. Once the teams are confident enough about the new architecture and the necessary knowledge is in place, then the organization can go ahead with the refactoring process for the existing functionalities from monolithic to microservice architecture.

The first decision to be taken during the refactoring process is to determine which functionalities could be the candidates to initialize the process. There are many factors that needs to be considered when deciding on the order of the refactoring process. The first step is to identify the standalone functionalities of the existing monolithic application. A single system might normally consist of several modules which could exist as separate standalone components. These will be good candidates to consider to initially to convert in to microservices. These standalone modules might be shared through the interface or these might be components that should function independently and not to be shared with other standalone components. The existing organization will already have knowledge on this area since the day to day operations might be built around these criteria. One example can be an ERP system which consist of manufacturing process handling, maintenance handling, finance, supply chain handling etc. Here customer order handling for the supply chain process could be taken as a single standalone functionality, since even though this is linked to other parts of the application through the interface, this will have its own underlying business logic and data.

After identifying the standalone modules inside the application, then it is necessary to identify from which module to start the transition process. One tip here to identify the most suitable functionality is to identify which functionalities could be changed more in the coming future. Since the changes could be easily made to the microservices, it would enhance the release of new additions quickly. The newly created microservices can be individually separately deployed and the consumers can give a specific feedback faster and this also enables the developers to try on new additions with the new architecture. The developers could experiment with the new feature and give a quality solution faster.

One more factor that can be considered when selecting the components to refactor is the technological advancements. If any component specially needs a technological enhancement to support the functionalities, then that could be considered as a good candidate for refactoring. After splitting the functionality and refactoring the monolithic application, these technical changes can be tried on the new microservice architecture. Since the microservice architecture supports implementations in multiple programming languages and technologies, it is a much better media to be experimented upon.

If there is a problem with the team composition or the structure in the existing organization that also can be taken in to account. If the teams are geographically separated and the communication is a common problem then when deciding on the new organizational structure, it can be best to divide the ownership of the microservices so that each team based on a single location handle a separate code base. This will minimize the communication problems and the ownership responsibility will ultimately lead to a much quality product as well. Even though this might be possible to achieve even through the monolithic architecture, it could be better applied for the microservices, also since the decisions can be taken independantly within the team in a specific location the decision making process also will be much faster and efficient which ultimately reduces the time for implementation.

The functionalities which has least dependency to the other modules is also a good candidate to be selected. The refactoring process will be easier since the tangled code segments are low or none. If any part of the application is highly cohesive and has minimum coupling with the other modules, these will obviously be a wiser choice as well. This will reduce the refactoring risk and the cost incurred, while the strong cohesiveness remains within the module and coupling is reduced.

The Domain Driven Design (DDD) is used to extract the candidates for the microservices from the monolithic application. Then the database schema needs to be compared with the selected candidate services and the inappropriate candidates which cannot be exposed as individual services need to be rejected.

Then it is necessary to select a communication protocol, data format and microservice framework. This technique is used to extract the domain specific services. These needs to be possible to operate independently.

3.1.4 Refactoring process - Refactor functionality

By the above techniques described in the previous section, after the candidate functionalities to be refactored is selected, and all the other business organization requirements are fulfilled then the transition from monolithic architecture to microservice can be started.

As the first step of the refactoring process, the monolithic application needs to be modularized to several components, which could be later decomposed in to several microservices. During this, it is a must to ensure high cohesiveness inside the modules. When these loosely coupled and highly cohesive modules are derived inside the monolithic application itself, it will clearly visible the interactions between these modules and thus the interfaces which will be applied to the microservices.

[42] The loose coupling and high cohesion could be achieved inside the modules through separation of concern. This means the basic business functionality of the module needs to be separated from the special handling code like synchronization. When too much code is wrapped around the same class and the purpose of the class becomes unclear it is possible to achieve this separation of concern through extract class process [43] The class that handles multiple responsibilities needs to be split up in to several classes which will handle a single responsibility. If the monolithic architecture code base is already in an architecture as Model- View- Controller architecture, then most probably these are already divided in to clear modules and the code is in a much better level.

To achieve high cohesion, it is possible to introduce various encapsulation strategies. If a method is used only inside the specific functionality and is not called by any other module this could be implemented as a private method or only with the package level visibility. Methods could be refactored along this line with hide method process, which enables to define a clear interface to the microservices.

3.1.5 Refactoring process - Refactor Database

When the functionality is successfully refactored and we have separate modules with highly cohesive nature with specially defined interfaces, the next thing to achieve is splitting at the database level. A database normally has intertwined dependencies. The same as we split up the standalone modules into different packages, this should be done the same at the database access level. The relational databases used in monolithic applications typically consist of entities with foreign key references and the context can irrelevant when accessing different database entities.

So with this nature database has somewhat of a nature of an integration point. But this is not supported for microservice approach and therefore the database access layer and the database structure needs to be refactored.

The process of selecting the data, preparing and extracting and transforming the data to transfer from one database to another is a quite expensive process. This migration process might not be a single migration since during the transition process the services could be changed and the boundaries between the services also could change, which results in multiple data migrations. One example of such changing scenario is, if a service identified as a candidate initially is too complex and needs to be split up in to different services the underlying data also needs to be split up again. Also if two services seems to perform a similar function, then these could be joined together along with the two databases. [44] So in these situations multiple data migrations are required.

To avoid the redoing of the data migration and to save the cost, it is possible to initially keep the data in the same database while the functionality is split up in to different services. Then after there is a stable set of services defined and the granularity is confirmed then the data migration process can start. This is a better approach since we could also benefit of the flexibility to finalize a stable set of services.

If two services were identified with excessive communications which introduce high coupling then these two services could be designed to merge as a one single service. The communication interaction between the selected services needs to be properly analyzed before concluding the set of selected services

Preventing Challenges during data migration

A common problem which occurs after data migration is the loss of transactional consistency. The techniques used by the monolithic application to ensure ACID transactions cannot be applied to the microservices in the similar manner, since the different microservices are now based on different databases. Therefore the developers cannot easily achieve the all or nothing transactional property in the similar manner as before. An example of such a transaction will be if a maintenance order is created for a vehicle object, based on the maintenance order a work order object needs to be automatically created. With the monolithic architecture this was easily achieved so both the operations occur in the single transaction and committed or the whole transaction will be rolled

back. When refactoring this functionality the developer needs to consider this transaction also into account. There are three ways how this could be achieved by the developer.

1) Eventual consistency

This is used in distributed computing and ensures that if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

2) Compensating consistency

Used to undo the work performed by a series of steps, which together define an eventually consistent operation, if one or more of the steps fail.

3) Distributed transaction

This is a database transaction in which two or more network hosts are involved. Usually, hosts provide transactional resources, while the transaction manager is responsible for creating and managing a global transaction that encompasses all operations against such resources.

All three options have advantages and disadvantages of their own and the approach that needs to be considered depends on the specific scenario. If some part of the transactions still resides in the monolith – for example if the work order handling still remains in the monolith and a microservice is created for the maintenance order – then some changes might need to be done to the monolithic application as well. Actually handling transactions in the microservice architecture could be harder than in the monolithic architecture due to these reasons. But for the success of this transition and changing the architecture of the application, it is critical to correctly define the transactions in the microservices and it is vital for the business operations.

Even if the microservice is fully separated from the monolith, this may still require to communicate with the monolithic application. Also two different microservices implemented might also need to interact with each other. For this interaction it is required to build an anti-corruption layer. The monolithic application could contain some coupling of data and data structures that we do not want to include in the microservices. To prevent this the anti-corruption layer will be of use.

Anti-corruption layer act as an in between adapter layer between the monolith and the microservices and offers the functionality for the microservice to access the monolithic application and vice versa. This also send information about any events occurred in the monolithic to the

microservice, if the microservices needs to take action upon those events. This can be a layer within the application or as an independent service. This will provide an interface to the monolithic application without any changes to the underlying business logic and provide an interaction path for the microservices. [45] This does not expose any underlying logic either and simply act as a gateway to access and communicate with the application.

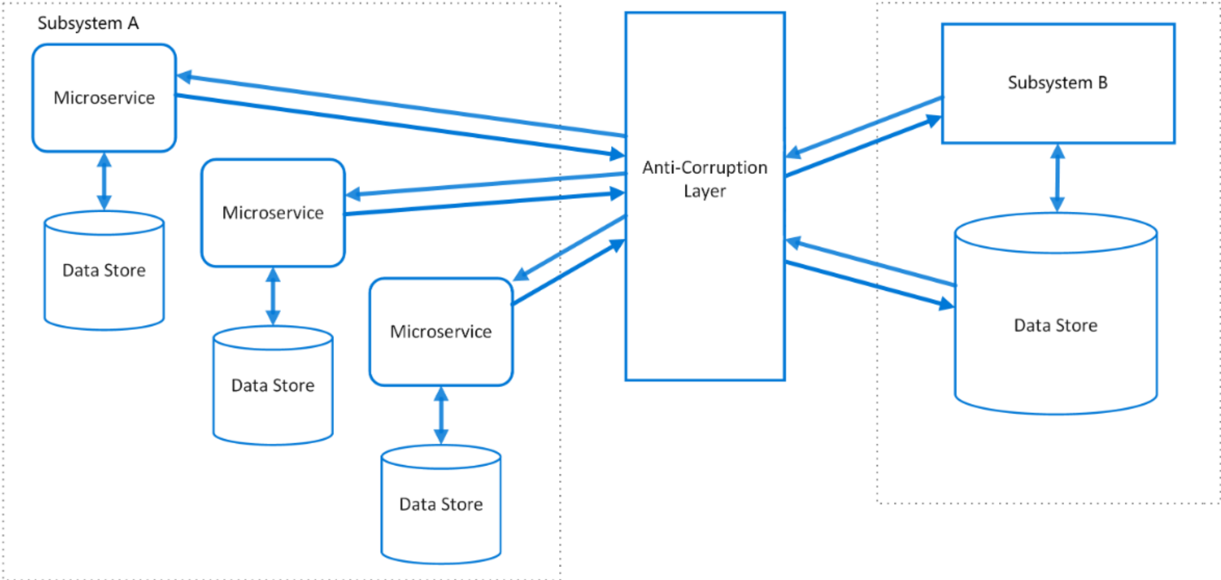


Figure 11 Anti-corruption layer between microservices and monilithic system

Figure11 shows the design with the anticorruption layer. The microservices need not consider the complicated interfaces and the data structures of the monolith because of the anti-corruption layer. These data structures might not be the best possible option for the microservice, therefore the anti-corruption layer present the data in a format which is ideal to the microservice. These services can be used by the microservice to send data to the monolithic or to receive data back. The anti-corruption layer might allow the microservice to use standard protocols like REST and specific communication protocols to the monolithic.

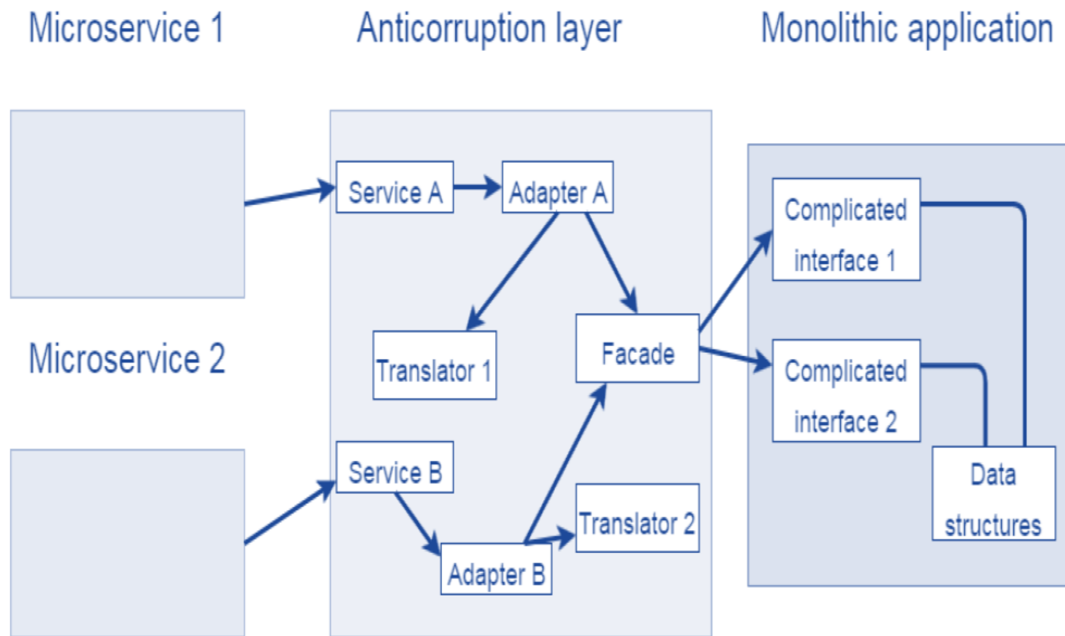


Figure 12 Underlying functions of the anti-coruption layer

As figure 12 shows, there are translators which handles the data received and sent between the two systems. This conversion of the data is a complex task and is specific function and that is the reason why this logic is separated from the adapters.

After the anti-corruption layer is implemented, it is now possible for the microservice to function itself as a separate service. The granularity of the service is correctly defined and all the other factors has been also considered and now a single microservice is generated. This same process needs to be repeated for the other remaining standalone solutions separately and for the new functionalities. The process of selecting candidates services also needs to be repeated again and after some time most of the code of the application will reside in microservices instead of the original monolithic application.

3.1.6 Methodology conclusion

This methodology provides an effective approach of migrating from a monolithic application to microservice based approach, while giving flexibility for the business organization to take time to educate on the new microservice architecture, space to make errors and learn about the new microservice architecture.

This way of keeping the monolithic code base parallel with the microservices for a period of time while the transition from one architecture to the next happens gradually is called strangler pattern. [46] The risk of the transition could be reduced to a minimum with this approach when compared to migrating the whole monolithic system to microservices at a one go. Also while this process is continuing, the already developed microservices could also get the benefit of short time to market for the new features while only the critical bugs needs to be handled in the monolithic application.

The transition process can be concluded once all the functionality in the monolith has been transitioned to the microservice or if the functionality that remains in the monolith has no use of transforming to a microservice. This could be some few functionalities which does not change frequently and trying with the transition could introduce unnecessary bugs and finally has no advantage over transforming to a microservice. Having the anticorruption layer between the monolith and the other microservices will be enough to handle this functionalities.

3.2 Evaluation Plan

The suggested solution through this research could be evaluated using two main approaches. That is through the API performance and the ability to scale through this architecture.

- API Performance

The performance of the SOA architecture proposed needs to be monitored in compared to the performance of the original system

- Ability to scale

The proposed architecture must be easy to scale with new functionalities, with minimal changes to the existing solution. This is one of the main advantages in the proposed solution. This level of scaling could be evaluated with the changing business needs.

CHAPTER 4

IMPLEMENTATION

The ultimate goal is to introduce an approach to convert an existing business application to micro service based architecture. The fairly complex and large systems could benefit from using microservices over monolithic architecture. But this transition has a number a challenges as well. A case study was created here to understand how this can be achieved and the underlying challenges. The case study is about the transition of open source software online shopping handling system of monolithic architecture to microservices. This process is carried out even when there are challenges in the transition since the benefits of having microservices will outweigh those challenges in the monolithic architecture.

This is practically applied by migrating one standalone component of the monolithic application to microservices and analyzing the time duration it has taken and the advantages of these microservices over the monolithic application also by considering the future developments in this functional area. Also a procedure on how a more dependent module can be converted to a microservice is also created. This is achieved after analyzing the dependencies between the different functional modules and finding the different steps needed for the transition. Based on these two scenarios, a very high level time estimate was deduced for the future transitions and the needed resources for the transitions. Because of the huge and complex code base, it is only possible to give a very high level estimate of the time needed for this.

Techniques for Implementation

An open source online shopping handling system is considered in this research as the case study to experiment with the transition from monolithic to microservice architecture. Here the step by step approach of refactoring a single module of the monolithic application to microservices is explained. This procedure can then be applied to the other remaining modules as well to complete the transition of the whole monolithic application to microservice.

As described in the methodology section, the first step of the process is to identify the standalone functionalities inside the monolithic application. It will not be practical and will be time consuming to identify all these functionalities initially. Instead of finding all at once initially it is possible to only select the most applicable functionalities first. That is the functionalities that is to be changed in the coming future, or the functionalities that could benefit from changing the technology to microservices or the functionalities which has the least dependency to the other functionalities. As the general guideline of this research any new functionality to be developed also has to be

developed as microservices, but the new developments were not considered during the practical implementation.

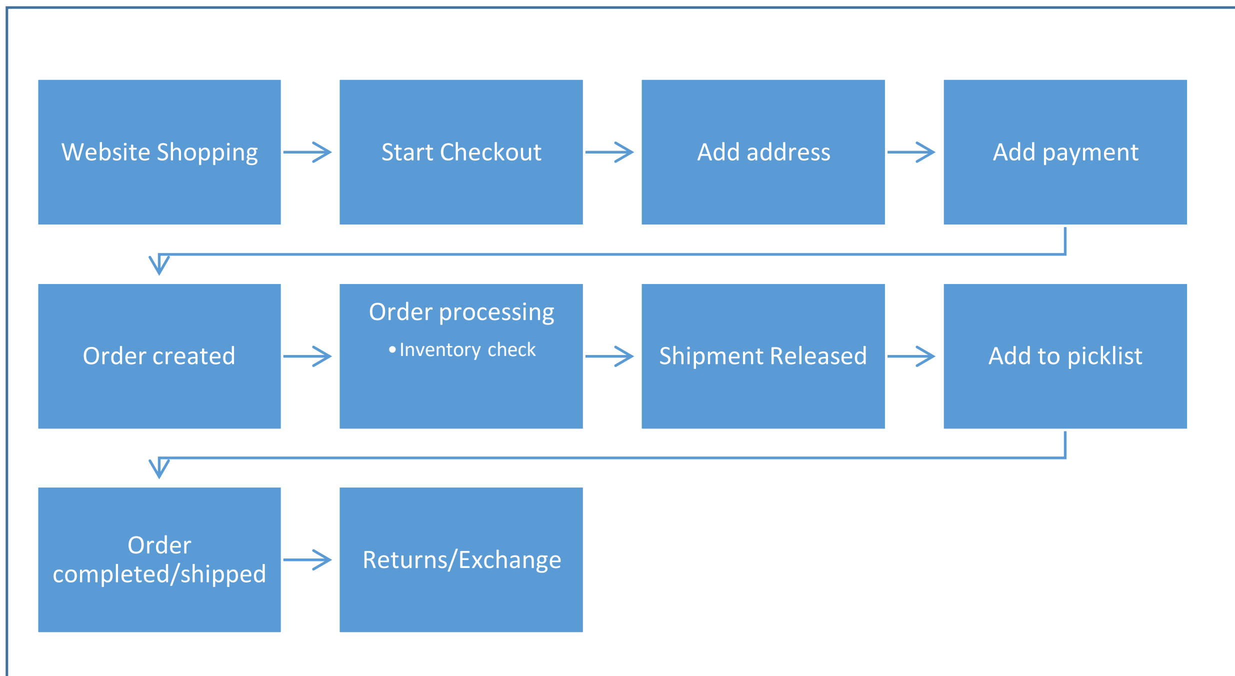


Figure 13 Extract services from the system [47]

Figure 13 shows the functionalities that could be extracted from the initial system which gives the overall flow of the business functionality.

The main business functionalities such as Registration to the online shopping system, shopping cart management, Payment Handling, Product management, Orders management, Shipping cost handling etc could be identified as business functionalities of this given system.

4.1 Identifying the candidate services from the monolithic system

Through Domain Driven Design candidates for services is identified.

In this implementation two different functionalities were selected Product management and Order Handling. These were selected based on the criteria described in the methodology section and these

two were selected since it could be used to get the maximum output for this research. Product management was selected because it has the lease dependency with the other functionalities and the code base is not much tangled with the other functionalities and since this will make the process of starting the implementation phase easy with the opportunity to gain more knowledge and experience on the new architectural style. The transition of Product management will be very useful and the application could have real benefits from it. The microservices advantages can be proved through this. Order Handling was selected to mainly present how to transform a complex and high dependent module of the application to a microservice. A guideline on how this can be achieved is presented here.

After these two were selected then the next step was to extract the code related to these two functionalities and separating this specific business functionalities from the rest of the application. It is also necessary to create automated tests for these two functionalities. If the monolithic application is already well defined in a modular manner as an example in MVC architecture, then this step can be skipped. But in most of the cases this step has to be carried out. But Product management functionality used in this research has a modular nature inside the monolithic application and hence this step was not necessary to perform which made the process easy and fast.

The functionality and the code base of Product management has been well tested which also made the migration process easier. These existing end to end, black box testing can be reused even after migrating to the microservice architecture to ensure the business functionalities have not changed during the transition.

Since the business requirement has not changed it was easier to convert these tests to go through the REST APIs.

After the code related to the Product management functionality has been transformed to the microservice, the next stage is to identify a plan for communication and interaction between the monolithic application and the newly created microservice. This initial communication with the monolithic application can be expressed as a must for every microservice created initially.

To handle this communication two approaches were considered. One is to create an anti-corruption layer between the monolithic application and the microservice and other one is to create a client between the monolithic application and the microservice.

If the communication between the microservice and the monolithic application is complex, then the anti-corruption layer must be created in order to achieve this. The data structures that need not be transformed from the monolithic application to microservices could exists and the anti-corruption layer allows to overcome these obstacles. This layer will help the microservice to get data and interact with the monolithic application and vice versa.

There are some situations where it is possible for the microservice and the monolithic application to directly call each other without the anti-corruption layer. Here the data structures need to be similar and the difference in the data models should not be much. In this scenario REST APIs and REST clients can be used to handle the communication between the monolithic application and the microservice. If it is needed to have communication both ways then it is needed to have REST APIs and REST clients for both the monolithic application and the microservice.

With Order Handling the anti-corruption layer is required because of the code that contains in the monolithic application. But when considering for Product management, it was not necessary to build the anti-corruption layer since the code base was in good shape in the monolithic application. REST API and REST client was able to create for Product management. This could be noted as a fact that if the data structures in the monolithic application is with good quality this could reduce the transformation time and make the process more effective since then it is not needed to create the anti-corruption layer which could be time consuming and a bit complex to create. Then the transition could be achieved by creating simple clients. But the developers need to consider this with caution since the anti-corruption layer will ensure that the data structures inside the application microservice will be of quality.

As part of the implementation it was also considered how this could be actually carried out in a business organization. One aspect was how to ensure the quality and the validity of the microservices. There are two techniques that could be used, that is canary development and feature flags. Even if the automated tests have been carried out still there is a chance that the transition could have introduced some bugs to the process. These bugs can be minimized by using canary development and feature flags in the business organization.

In canary development the microservice is at first released to a small set users so the impact of the microservice will be limited to a small number of users. The microservice will be heavily monitored during this time and possible error situations are noted down. This procedure is much better since any error on the microservice will only affect a small number of users. After ensuring smooth running of the microservice, then more users can be introduced and finally all the users will be using the microservice.

Along with the canary development, feature flags can also be used when doing the final transformation. Feature flags needs to be created in the monolith application to handle the final transformation. The functionality that is already transformed to the microservice must be wrapped in the feature flag in the monolith application. Then this feature flag can be used to disable that specific function from the monolithic application and use the microservice instead. The benefit that could be gained here is that if the microservice gives any error or does not behave as expected, then the possibility is there to disable it and enable the monolithic feature back on. After issues with the microservice is fixed and smooth operation of the microservice can be ensured, then the feature can be turned off from the monolith and the relevant code and the feature flag can be removed from the monolith. Now it can be safely said that the transition of the microservice is complete.

4.1.1 Proof of concept

The proof of concept here is for Product management and Order Handling of the business functionalities. For Product management the monolithic functionality was transitioned to the microservice and for Order Handling a guideline was created how this can be achieved, since due to the high dependencies in the code it is not possible during the scope of this project to complete the transition. In here the guideline is given as a general guideline on how the high dependent business functionalities could be transformed to microservices. However the main focus in based on these two business functionalities, and how these two could be transitioned from the monolithic to microservice architecture.

Business organizational challenges were not taken as part of the scope and only the technical challenges were addressed in here. On top of these organizational challenges, there will be more challenges in the production environment such as monitoring the microservice execution and logging etc. These were not considered in this implementation since only the architectural side of the transition was considered in here and not the business organizational and operational side.

Product management proof of concept

The transition was started from this stage since Product management is loosely coupled and has a modular structure inside the monolithic application. Also the test coverage was at a good level which helps more in the transition process. Starting first from a simpler one reduces the risk of failure and gives more knowledge and experience about microservices. The experiences gained from the simple transitions can be useful when it is necessary to transform complex functionalities.

MySQL is the relational database management system that was chosen. This is because the monolithic database which already exists is a MySQL database and that was of good quality.

The Product management is divided in to controller, service and data access layers. The business logic of the service reside in the service layer and the controller and data access layers are thin. In the controller layer the microservice provides an OData API for the consumers of the microservice. The API provides CRUD (create, read, update and delete) functionality along with querying functionality. Because of the simple API the complexities of the underlying business logic is hidden from the end users. The .svc provides a clear interface to the business functionality making the functionality loosely coupled from the other business functionalities.

4.2 Protocols used for microservices

The http based REST services are created using OData protocol which allows to uniquely identify a resource using Uniform Resource Identifiers(URIs). This could be applied in many scenarios including CRM systems, for traditinal websites, file systems, for CRUD operations on a relational database etc.

One of the many reasons for selecting OData service over basic REST service is since it supports various kinds of query options for querying data.

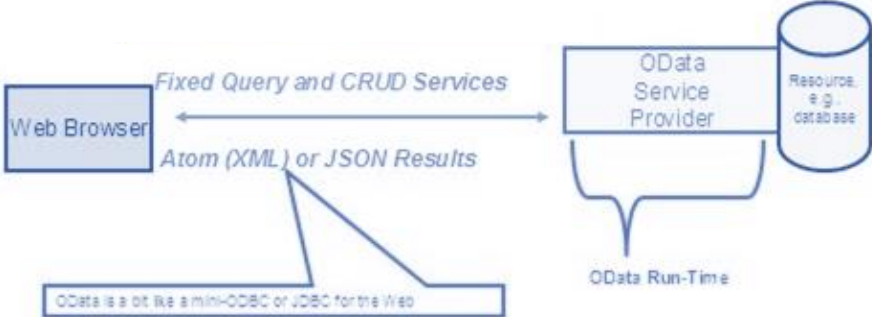


Figure 14 A single microservice

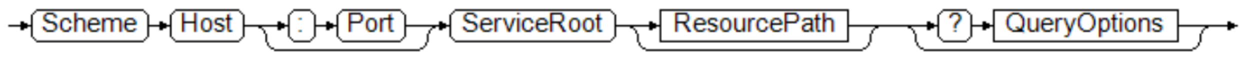
The figure 14 illustrates a single microservice exposed in this approach. [48]

The resources could be viewed in two formats, XML based ATOM format and the JSON format. A set of entries is called a collection and collections are represented as Atom feeds.

```xml
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<service xml:base="localhost/../Products.svc/" >
  <workspace>
    <atom:title>Default</atom:title>
    <collection href="Products">
      <atom:title>Products</atom:title>
    </collection>
    <collection href="Categories">
      <atom:title>Categories</atom:title>
    </collection>
    <collection href="Suppliers">
      <atom:title>Suppliers</atom:title>
    </collection>
  </workspace>
</service>
```

Figure 15 OData service details from .svc

The figure 15 shows the .svc file of an OData service. OData service can hosted on a web server and exposed through the Web or a local network via an .svc file. This is the entry point to the service.



The following are two example URIs broken down into their component parts:

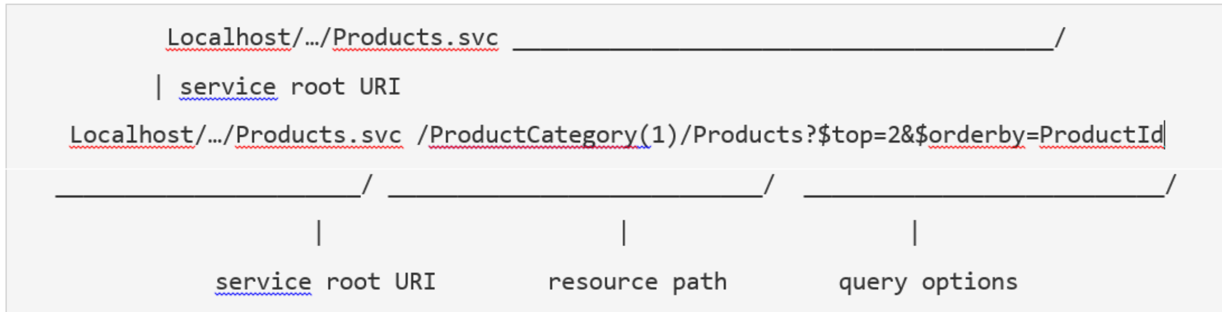


Figure 16 OData service request

Figure 16 shows an OData service request with different elements which helps to request for the resource.

Here, the entries are represented as a set of Atom elements and with a listing of the properties.

The service root identifies the root of the OData service. The resource to be identified here needs to be an AtomPub service docuement or JSON based service document. All the collections will be listed down in the service document enabling the clients to easily identify them.

The resources for whom the CRUD operations will be executed upon will be identifies in the URI through the resource path section.

4.2.1 Service Metadata Document

The organization and the structure of all the resources are described in the Service MetaData Document. The resources will be displayed as HTTP end points. These are described in Entity Data Model (EDM) terms. Entities and associations between two or more entities, entities are grouped as entity sets.

```
<EntityType Name="Product">
<Key>
<PropertyRef Name="ID"/>
</Key>
<Property Name="ID" Type="Edm.Int32" Nullable="false"/>
<Property Name="Name" Type="Edm.String" m:FC_TargetPath="SyndicationTitle" m:FC_ContentKind="tex
<Property Name="Description" Type="Edm.String" m:FC_TargetPath="SyndicationSummary" m:FC_ContentK
<Property Name="ReleaseDate" Type="Edm.DateTime" Nullable="false"/>
<Property Name="DiscontinuedDate" Type="Edm.DateTime"/>
<Property Name="Rating" Type="Edm.Int16" Nullable="false"/>
<Property Name="Price" Type="Edm.Double" Nullable="false"/>
<NavigationProperty Name="Categories" Relationship="ODataDemo.Product_Categories_Category_Produc
<NavigationProperty Name="Supplier" Relationship="ODataDemo.Product_Supplier_Supplier_Products"
<NavigationProperty Name="ProductDetail" Relationship="ODataDemo.Product_ProductDetail_ProductDe
</EntityType>
<EntityType Name="FeaturedProduct" BaseType="ODataDemo.Product">
<NavigationProperty Name="Advertisement" Relationship="ODataDemo.FeaturedProduct_Advertisement_A
"FeaturedProduct_Advertisement"/>
</EntityType>
<EntityType Name="ProductDetail">
<Key>
<PropertyRef Name="ProductID"/>
</Key>
<Property Name="ProductID" Type="Edm.Int32" Nullable="false"/>
<Property Name="Details" Type="Edm.String"/>
<NavigationProperty Name="Product" Relationship="ODataDemo.Product_ProductDetail_ProductDetail_P
</EntityType>
<EntityType Name="Category" OpenType="true">
<Key>
<PropertyRef Name="ID"/>
</Key>
<Property Name="ID" Type="Edm.Int32" Nullable="false"/>
<Property Name="Name" Type="Edm.String" m:FC_TargetPath="SyndicationTitle" m:FC_ContentKind="tex
<NavigationProperty Name="Products" Relationship="ODataDemo.Product_Categories_Category_Products"
</EntityType>
<EntityType Name="Supplier">
<Key>
<PropertyRef Name="ID"/>
</Key>
<Property Name="ID" Type="Edm.Int32" Nullable="false"/>
<Property Name="Name" Type="Edm.String" m:FC_TargetPath="SyndicationTitle" m:FC_ContentKind="tex
<Property Name="Address" Type="ODataDemo.Address"/>
<Property Name="Location" Type="Edm.GeographyPoint" SRID="Variable"/>
<Property Name="Concurrency" Type="Edm.Int32" ConcurrencyMode="Fixed" Nullable="false"/>
<NavigationProperty Name="Products" Relationship="ODataDemo.Product_Supplier_Supplier_Products"
</EntityType>
<ComplexType Name="Address">
<Property Name="Street" Type="Edm.String"/>
```

Figure 17 OData metadata document

Figure 17 shows an example metadata document which describes the resources exposed through the service. The order resources could be identified as below.

OData Resources as described in an Entity Data Model by

- Collection :- Entity Set (A navigation property on an entity type that identifies a collection of entities)
- Entry :- Entity Type (Note: Entity Types may be part of a type hierarchy)
- Property of an entry :- Primitive or Complex Entity Type Property
- Complex Type :- Complex Type
- Link :- A Navigation Property defined on an Entity Type
- Service Operation :- Function Import

An OData service to function correctly it is required to always have a usable and stable EDM, since there will not be validation of the EDM by the OData library since this might impact the performance.

Meta data is defined in the EDM provider classes. These differentiate the runtime EDM object instances from the originally defined OData EDM objects in the code.

4.2.3 Handling various operations through the services.

Here the service operations much like the same way as representing for resources will be identified through the URI. Functions could be exposed through the service with input parameters of primitive type and the return value of a function could be single or a collection of primitive or complex type, a single or a collection of entries.

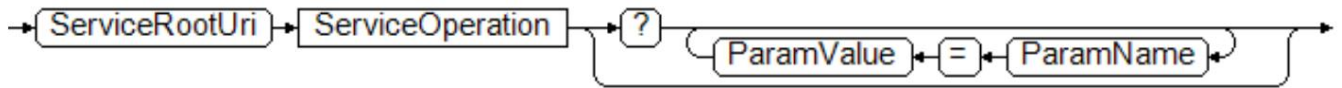


Figure 18 Functions exposed through services

As figure 18 shows the business function is exposed through the service as a service operation and the order in which the parameter values are passed is unimportant as long as the parameter names are correctly defined.

OData protocol implementation could be hanlded using many different libraries which are available for different languages. Apache Olingo is a Java library that implements the OData protocol.

HTTP methods GET, PUT, POST, PATCH and DELETE are used for the primitive CRUD (create, read, update and delete) operations. In here the OData request is handled in a single call call to a processing method. This means the implementation has to read collections, their relations and functions and actions may all occur before a simple command is executed in the request URI.

The following code shows an example of an OData API. This shows all the main entry points to the service. The entity sets that are exposed through the API along with functions and actions can be seen in here.

The OData Protocol is different from other REST-based web service approaches in that it provides a uniform way to describe both the data and the data model.

```
GET http://localhost:8080/OnlineShoppingSystem/ProductManagement.svc/Products?$top=2 &amp; $select=ProductId, ManufacturerName &amp; $filter=Sales/any(d:d/Budget gt 3000)

OData-Version: 4.0
```

Figure 19 OData GET request with Query parameters

Figure 19 illustrates an OData request consist of query strings. OData could define a series of system query options that can help you construct complicated queries for the resources to be loaded. This example request shown in figure 11 is to query for first 2 products in the system which has been included in atleast one sale that costs more than 3000, and the query needs to display only the Product ID and the Manufacturer Name.

```
POST http://localhost:8080/OnlineShoppingSystem/ProductManagement.svc/Products/

OData-Version: 4.0

Content-Length: 428

Content-Type: application/json

{

   'ProductId':'N545-500',

   'ProductDescription':'Asus X510 i5/4G/1TB/2GB/W10',

   'ManufacturerName':'Barclays',

   'Price':'130000',

   'ProductRevision':[

      '23TrJ76NjtGuI'

      '8UJKMNuen4556H'

   ],

   'AddressInfo':[

      {

         Address: '55 1st street colombo.',

         City: {

                        District: 'Colombo',

                        Lane: '1 lane',

                        Town: 'Kottawa'
```

```
        }
      }
   ],
   'ProductCategory': 'Laptops',
   'ProductUpdatedDate': '2018-10-03'
}
```

Figure 20 Creating a new resource through OData request

As figure 20 shows when creating a new resource through an OData request, the client needs to send a POST request containing a JSON representation to the server.

A good documentation is needed for the microservices for the other developers to easily communicate with from other services. OData meta document provides this functionality

For the newly created client to communicate with the monolithic application it is necessary to create a new client that could handle these requests. Since the code base in the monolithic architecture was in good condition, it is not necessary to create a separate anti-corruption layer in this scenario. A simple client will do the job here.

Transformation plan for Order Handling

The next stage is to define a plan to transform highly dependent nature of the modules to microservices. Here the business contect chosen is Order Handling. Due to the time constraints in the research and the complexity of the code base, in here only a plan on how to handle these situations are explained.

Since the whole transformation can be very long it might not be feasible in the financial aspects to carry this out in one go. Therefore a much wise choice is to split the transformation process in to small pieces. Each individual step of the process will add value to the organization and will make the complex code base a bit easier to change.

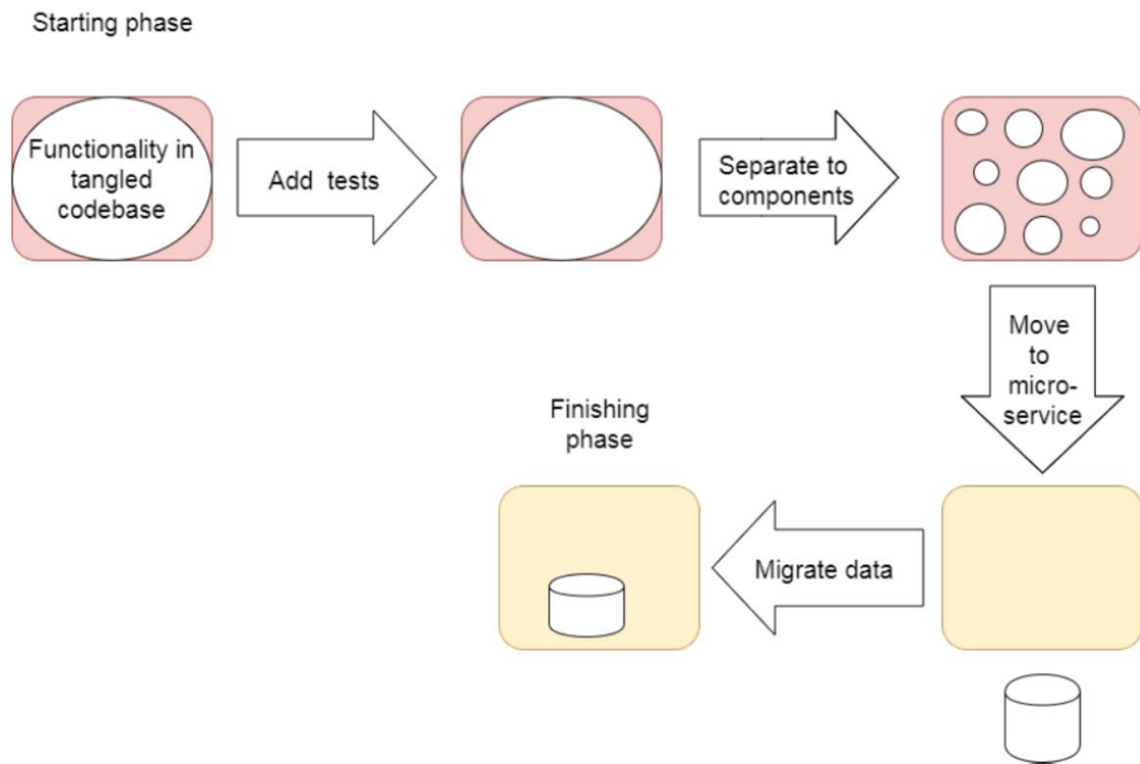


Figure 21 Transition process

In figure 21 the rectangle shows the phases of the transition. In the first 3 stages, the functionality still resides in the monolithic application. In the last 2 stages shown in the figure the monolithic application is created.

The first step here is to add automated test coverage if any are missing. As previously mentioned there must be strong unit tests created which are fast to give a highly accurate feedback to the developers. These unit tests needs to be backed with integration testing and end to end testing to ensure that there are no regression issues created due to the transition.

When it is confident that the testing level is at an acceptable level, then the process of separating the functionality inside the monolith can be started. This in detail means creating multiple classes if there are complex classes with too many responsibilities, also using programming techniques to ensure high cohesiveness and low coupling. Each individual component should focus on one business functionality and encapsulate the data and functions inside that component.

It is possible to achieve this kind of architecture in different means by different programming languages. For example in Java by using visibility modifiers and packages. If all the classes in the module are public that implies that they can be accessed from the whole module and will eventually lead to low cohesion and high coupling. To prevent this, instead of the public classes, it is possible to derive protected classes. Then when used with the packages, these classes can only be accessed inside that separate package. This will add a barrier to low quality code and will provide the microservice with a very clear interface to the business functionality.

The next step is to take the selected modular component, extract it from the monolith and create it as a separate microservice. As with the previous case, once the business functionality is in a well-structured modular format inside the monolith, it is not very challenging to transfer it to a microservice. Now only the migrating of the code base has been done and the database migration still remains. Microservice is still using the database of the monolithic application. Then before doing the data migration it is necessary to confirm on the service boundaries of the microservice and the granularity of the microservice. If these are not cleared, the data migration process might have to be done multiple times which involves a high cost. Therefore once these are confirmed, it is possible to start with the data migration. Also it is not a good decision to wait with this last phase for too long, since this final transition at the data migration will eventually enables scalability of the microservice, separate deployment of the microservice and implementation using multiple programming languages. After the data migration process is completed, before transforming the full traffic to the microservice, canary deployments and feature flags can be used to deploy and test the microservices in production. Also as discussed previously, the anti-corruption layer might have to be created.

By using Java visibility modifiers and packages it is possible to create separate packages for different CRUD operations for different functions. By doing this, it is possible to achieve separation of concern inside the monolith itself. But this needs to be done with caution, since just changing the visibility modifiers could have a negative effect if not done after proper analysis. Otherwise this could break the existing architecture. But when transitioned a particular business context into a microservice, it is lot harder to break.

The current code situation has many issues. One is God Class [49]. A God class can be defined as a class which contains too many responsibilities and this will be an obstacle during the refactoring.

God classes are hard to modify and harder to understand. If the God class has business logic together with the data access logic, this could also lead to bugs during the transition. It is a must to always separate the business logic from the data access code and for the God classes, they need to be divided into smaller classes which have well defined business function instead of one complex class packed with all the functionalities.

Before the refactoring in to smaller methods, it is better if some unit tests could be created to test these functionalities. When the unit tests are created using the java visibility modifiers, it is possible to move the functionality into separate classes and packages. Again it could be suggested from here, at this stage to use different refactoring tools that different Java IDEs provide to reduce the manual that could error in this process as much as possible. The final goal of this stage is to make the code highly cohesive and loosely coupled.

After packages have been created with limited to the selected business function and the interface is clear, then it is possible to call this architecture as a modular monolith [50] This could be explained as similar to a component based architecture since it has loose coupling, high cohesion with separation of concerns. [51] Now the microservice functional decomposition will be different to the monolith, since in the monolith all are still running inside the same process and in the microservice each process in implemented in a separate microservice. Microservices also provides easier upgrades and replacements. Also the modular monolith makes the transition to microservices even easier. This was already the case with Product management. When doing the transition it is not advised to try to get it done in one go, but take small steps in the process in order to prevent regression issues.

After the God class is removed and packages are created for each individual business context, inside these packages only the classes that should be accessible from the other packages will be set as public and the other classes need to be set as protected. Business logic resides in the service layer and the services communicate with each other if they need to interact with each other in the service layer.

The next step is to transform these processes to microservices. All the processes does not need to be transformed at once to microservices. The regression bugs can be minimized and more experienced could be gained when the transition is carried out one by one. It will be better to start

from the modules which are updated frequently or can be benefited from other advantages of microservices.

After the transition to microservices, an anti-corruption layer could be built between the monolith and the new microservices. This anti-corruption layer will convert the data structures that are inside the monolith to the data structures of the newly created micro service and vice versa. After the whole functionality has been transformed to microservices, the anti-corruption layer can be removed since the communication between the monolith and the microservices is not needed anymore.

Using this plan mentioned above can be used as a general approach to transition towards microservices. The main challenges here could be identified as converting a monolithic application to component based approach and the organizational and operational challenges that could be introduced with this approach. But these are solvable with the correct resource and time.

The metadata document of the microservice could be invoked from the following URL

<serviceroot>/$metadata

The service document can be invoked via the following URI.

<serviceroot>/

```
http://localhost:8080/OnlineShoppingSystem/ProductManagement.svc/Products
```

```
{
  "@odata.context": "$metadata#Products",
  "value": [
    {
      "ID": 1,
      "Name": "Notebook Basic 15",
      "Description": "Notebook Basic, 1.7GHz - 15 XGA - 1024MB DDR2 SDRAM - 40GB"
    },
    {
      "ID": 2,
      "Name": "1UMTS PDA",
      "Description": "Ultrafast 3G UMTS/HSDPA Pocket PC, supports GSM network"
    },
    {
      "ID": 3,
      "Name": "Ergo Screen",
      "Description": "17 Optimum Resolution 1024 x 768 @ 85Hz, resolution 1280 x 960"
    }
  ]
}
```

Figure 22 The data viewed through the microservice

Figure 22 shows the data queried from the ".svc/Products" query. This will list down all the products in the repository. OData filtering could be added here to optimize the results.

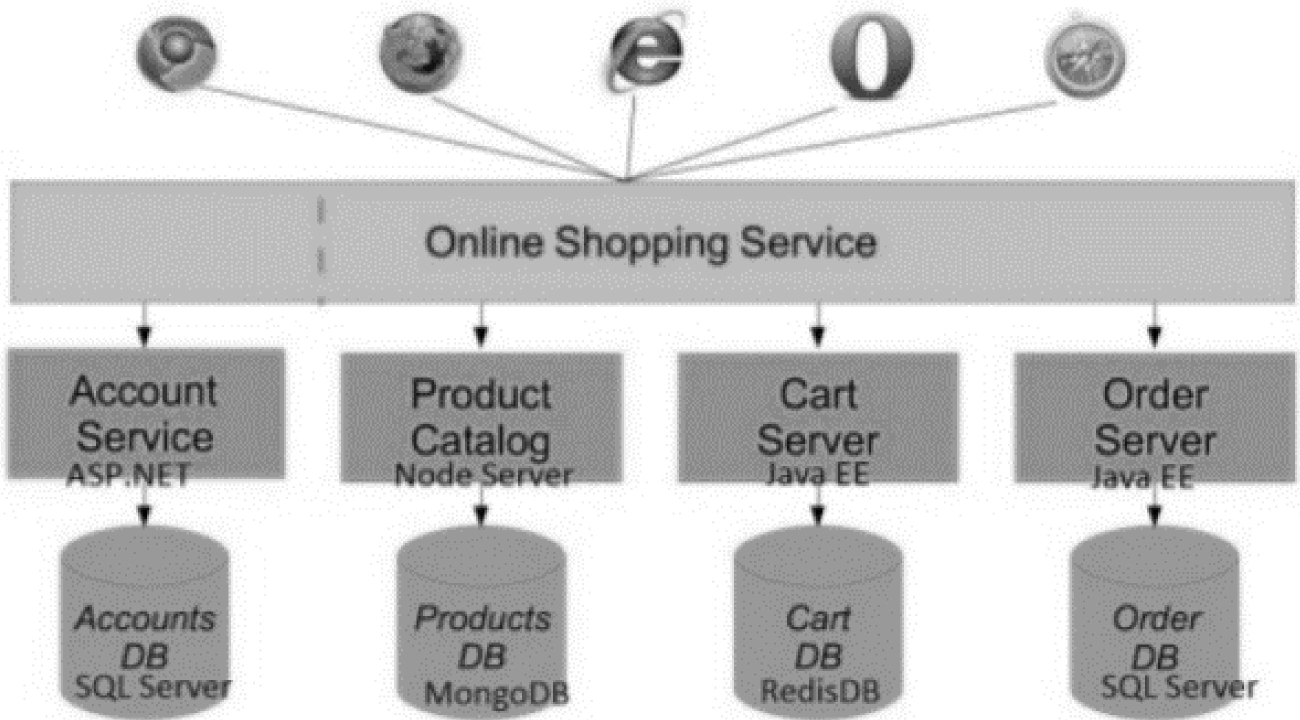


Figure 23 Basic service oriented architecture derived from the research

Figure 23 shows the basic architecture of the online shopping system with the microservices implemented.

OData service implementation can be derived from CsdlAbstractEdmProvider class. EDM objects can be created from overriding the methods of this abstract class. EDM model basically presents the EntityTypes available and the references between entities.

```
28   */
29  public abstract class CsdlAbstractEdmProvider implements CsdlEdmProvider {
30
31    @Override
32    public CsdlEnumType getEnumType(final FullQualifiedName enumTypeName) throws ODataException {
33      return null;
34    }
35
36    @Override
37    public CsdlTypeDefinition getTypeDefinition(final FullQualifiedName typeDefinitionName) throws ODataException {
38      return null;
39    }
40
41    @Override
42    public CsdlEntityType getEntityType(final FullQualifiedName entityTypeName) throws ODataException {
43      return null;
44    }
45
46    @Override
47    public CsdlComplexType getComplexType(final FullQualifiedName complexTypeName) throws ODataException {
48      return null;
49    }
50
51    @Override
52    public List<CsdlAction> getActions(final FullQualifiedName actionName) throws ODataException {
53      return null;
54    }
55
56    @Override
57    public List<CsdlFunction> getFunctions(final FullQualifiedName functionName) throws ODataException {
58      return null;
59    }
60
61    @Override
62    public CsdlTerm getTerm(final FullQualifiedName termName) throws ODataException {
63      return null;
```

Figure 24 CsdlAbstractEdmProvider class [37]

The class file in figure 24 could be overridden to provide the EDM document to the service

Some examples from the EDM document

- ***getEntityType()***

  Here we declare the EntityType "Product" and a few of its properties

- ***getEntitySet()***

  Here we state that the list of products can be called via the EntitySet "Products"

- ***getEntityContainer()***

  Here we provide a Container element that is necessary to host the EntitySet.

- ***getSchemas()***

The Schema is the root element to carry the elements.

- ***getEntityContainerInfo()***

Information about the EntityContainer to be displayed in the Service Document

```
public CsdlEntityType getEntityType(FullQualifiedName entityTypeName) {

  // this method is called for one of the EntityTypes that are configured in the Schema
  if(entityTypeName.equals(ET_PRODUCT_FQN)){

    //create EntityType properties
    CsdlProperty id = new CsdlProperty().setName("ProductId").setType(EdmPrimitiveTypeKind.Int32.getFull
QualifiedName());
    CsdlProperty name = new CsdlProperty().setName("ProductCategory").setType(EdmPrimitiveTypeKind.Stri
ng.getFullQualifiedName());
    CsdlProperty  description = new CsdlProperty().setName("ProductDescription").setType(EdmPrimitiveType
Kind.String.getFullQualifiedName());

    // create CsdlPropertyRef for Key element
    CsdlPropertyRef propertyRef = new CsdlPropertyRef();
    propertyRef.setName("ProductId");

    // configure EntityType
    CsdlEntityType entityType = new CsdlEntityType();
    entityType.setName(ET_PRODUCT_NAME);
    entityType.setProperties(Arrays.asList(id, name , description));
```

```
    entityType.setKey(Collections.singletonList(propertyRef));



    return entityType;

  }
```

Figure 27 Entity type property overriden

As shown in figure 27 all most all the methods provided here has a parameter of type FullQualifiedName, which provider object instance needs to be returned.

EntitySet is a crucial resource when data is requested through OData. Entity sets and global actions and functions are the main entry point to the OData service

When accessed for
http://localhost:8080/OnlineShoppingSystem/ProductManagement..svc/$metadata

```
<EntityType Name="Product">
        <Key>
          <PropertyRef Name="ID"/>
        </Key>
        <Property Name="ProductId" Type="Edm.Int32"/>
        <Property Name="ProductName" Type="Edm.String"/>
        <Property Name="ProductShortDesc" Type="Edm.String"/>
      </EntityType>
      <EntityContainer Name="Container">
        <EntitySet Name="Products" EntityType="shopping.Product"/>
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

Figure 25 Metadata document for Product resource

CHAPTER 5

EVALUATION AND CONCLUSION

In this chapter the evaluation of the implementation and the final conclusion of the research will be presented. The world load of the transition for Product management is discussed along with the challenges the transition provided. Also some of the biggest challenges that could arise in this plan for Order Handling transition process will also be presented in this chapter.

5.1 Evaluation

The time taken to implement the Product management microservice was measured. It was taken around 4 weeks of development time for the developer. This can be used as a very rough estimate since for the future developments, because each and every functionality is different and the complexity also differs with the business functionality. Due to these reasons some transitions can take a shorter time while some others might take longer time. Since this is an incremental approach, the experience gained from a previous stage and the lessons learned could be benefitted to the latter stage. Also it has to be taken into account the time to optimize and test the microservices implemented in the testing environments and in the production environment.

To ensure proper service in the production environment, a container orchestration system for automating application deployment, scaling and management such as Kubernetes or any other similar container orchestration is required.

Building the aggregate logging and monitoring takes some time. But since these are similar kind of tasks for all the services, once these are created it is easy to duplicate them for the other microservices.

Together with these tasks it is required to build Continuous Integration/Continuous Deployment (CI/CD) pipeline for each microservice. This was not implemented as proof of concept during the implementation of this research. But when taken into estimates, this can be taken as a smaller task than the codebase transformation. It is possible to duplicate these pipelines for other microservices.

The complexity of actually running multiple microservices in the production environment cannot be clearly evaluated from this procedure. In here only one microservice is considered for analysis and the challenges that comes with distributed computing is not addressed.

From this it is possible to come to a conclusion that transition of a functionality which is already in modular architecture inside the monolithic application to microservice might not propose a high risk. But on the other hand it does not imply that the transition is easy. The biggest challenge is not individually transforming the microservice or running it independently but the challenge is running the microservice with multiple other service interactions in the production environment. Also before transitioning from monolith to microservice architecture, converting the monolithic to modular monolithic will make the process much easier. Also one more noticeable point is that is easier for the developers to grasp the business functionality once the transition is done and the individual services are separately exposed through microservices. Since the microservices can be deployed separately, it is possible to make faster releases as well. Using CI and CD pipelines which are enabled for the microservice architecture will help to boost the faster releases. Also the testing that needs to be done before releasing a microservice to the production is lower than the testing that has to be done with the monolith.

Microservices also enable development through multiple programming languages (polyglot implementations), and this introduces the possibility to select the most suitable the programming language for the job instead of implementing all the functionality in one common language due to the nature of the application.

The reliability of the result depends on what are the factors that are considered to measure the reliability. It is possible to consider two main factors. The first factor is the service of a single microservice and the second one is whether the process functions well when the whole monolithic application is transitioned to microservice.

When a single service is transitioned from the monolithic to microservice, it is easy to achieve the high cohesiveness and low coupling. This could be proved from the actual implementation of Product management which was smooth without any critical issues. So in that angle it is much better to go ahead with microservice approach. But when the whole complex system is considered, there is no guarantee initially, that the full microservice architecture will remain one way or the other. It requires repetitive continuous planning and incremental developments. Developers needs to be trained, and creating libraries and tools to support the developers will also require time and needs to be planned properly in order to achieve success in microservice architecture.

5.1.1 User survey to evaluate microservices

Evaluation Criteria	Monolithic	Microservice
Deployment	Deployment is complex since the whole system needs to be deployed at once. This required a system down time to be announced which will affect the ongoing work and also time might need to be reserved for maintenance.	The deployment of microservice is much easier and simple, since they can be deployed individually without any down time or with a very small down time.
Programming language dependency	Normally the whole application is built on a single programming language	Since the underlying technology doesn't matter as long as the microservice interface is clear and stable, different programming languages can be used to develop different microservices.
Scaling	When expanding the application with regard to some criteria, it is necessary to expand the whole application, even if the limitation is only at a single point.	Individual microservices can be scaled independently without any change to the other microservices.
Codebase	The whole application is based on a single code base.	There will be separate code bases for each microservice

Maintainability	Due to the large size, it is often hard to understand and maintain.	Since each microservice is normally maintained by a specific team and the code bases are separated, it is easy to understand and maintain.

Table 1Evaluation Criteria for microservice vc Monolith

From the table 1, it is clear that when most of the criteria are considered, microservices seems like a better choice over monolithic architecture.

A user survey was carried out to further analyze these criteria with multiple developer's perspective. Software developers with varying experience levels were targeted for this survey to get an accurate an unbiased opinion. Figure 26 shows the results of this analysis.

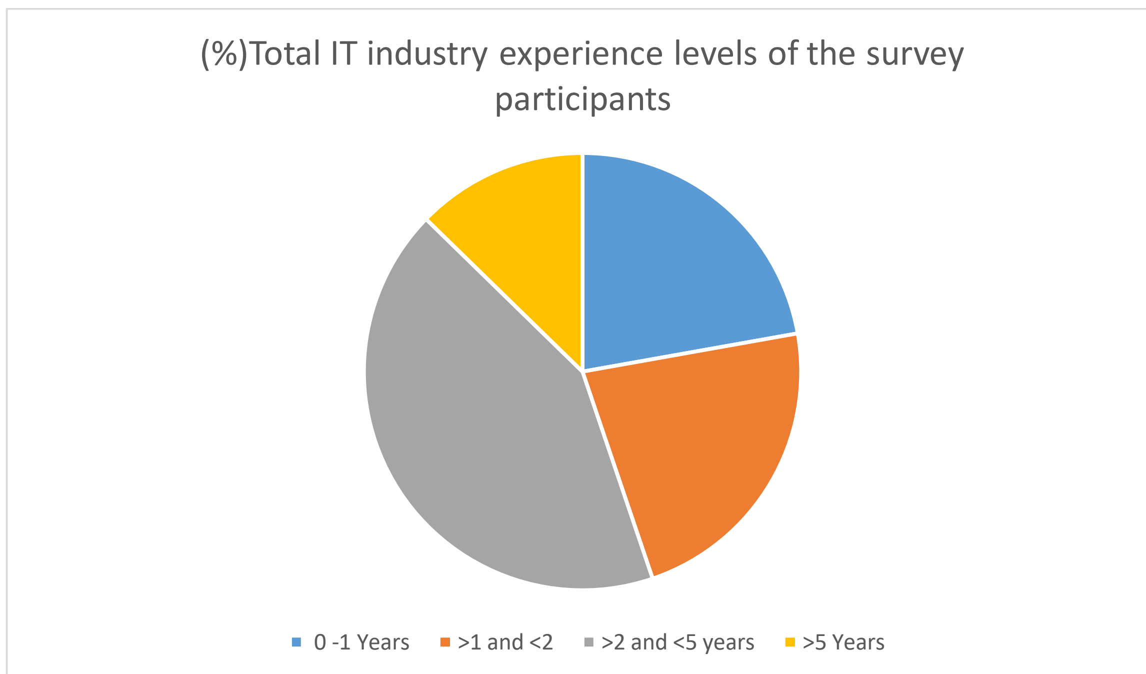


Figure 26 Experience levels of the participants

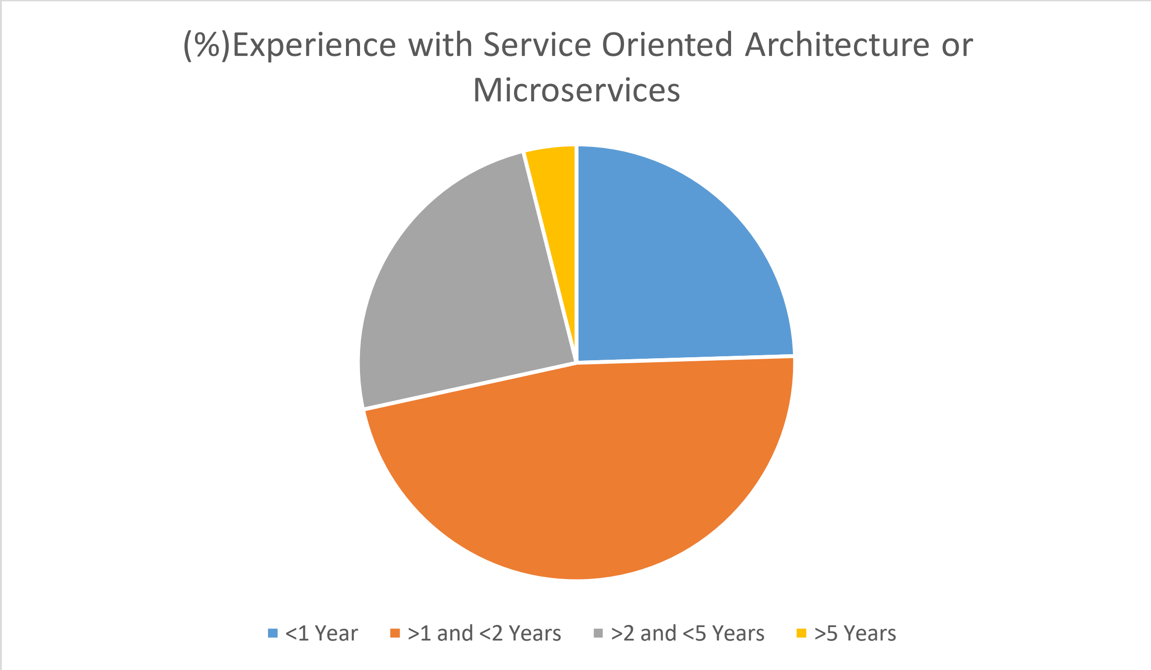


Figure 27 Experiance with SOA or microservices

Based on the initial background gathering data, it was decided that the experience and the knowledge of the software developers responded to the survey was enough to get a rough evaluation on the topic. The experience of participants specially in the area SOA is in figure 27.

<u>Benefits of Microservice usage with the importance</u>

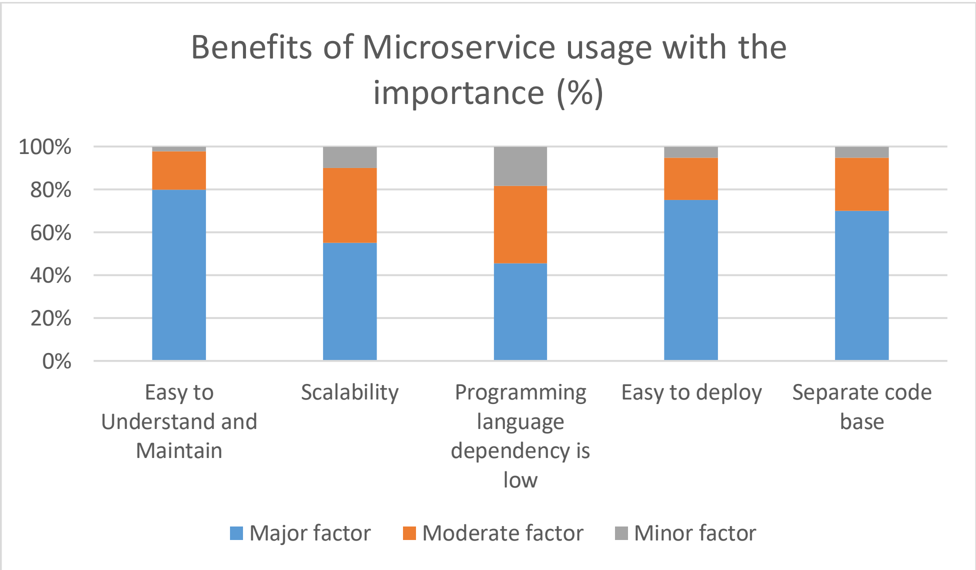


Figure 28 Benefits of Microservice usage with the order of importance (%)

As the figure 28 shows from the responses to the survey, the users perspective is that the easy to understand and maintain, easy to deploy and having a separate code base is major benefits when considering moving from monolithic to microservice architecture. One more clear observation that can be seen here is that, for the all the categories more than 50% consider those as major benefits of microservices over monolithic architecture. Programming language dependency is low has been seen as a major factor only by 40% of the participants.

<u>The importance of challenges with regard to microservices</u>

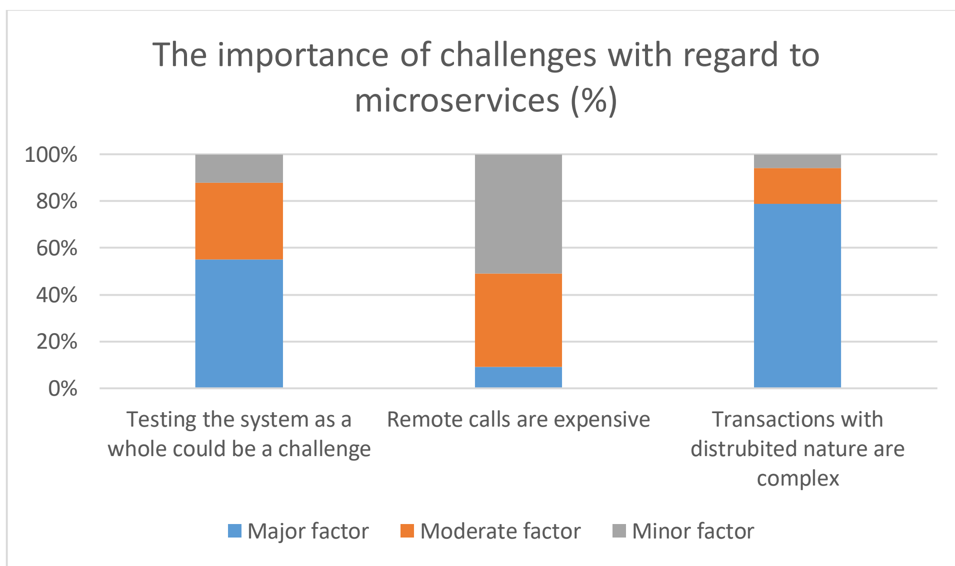


Figure 29 The importance of challenges with regard to microservices

As the figure 29 shows to understand the users perspective on the importance of challenges with regard to microservices is included in the survey. The three main areas that were analyzed here is the challenge in handling transactions with distributed nature, Handling expensive remote calls and the process of testing the whole system at once.

From the user's point of view, the most important challenge is handling transactions with distributed nature. This is a complex task which could be considered as a major challenge factor. 55% of the users also responded think that testing the whole system is also a major challenge factor. Also one interesting factor that can be noticed from the graph is that only 10% of the users believe that expensive remote calls are a major challenge.

This survey results can be taken as a general opinion on the microservices. But the validity may be questioned, since the survey is limited to only small number of users of microservices. This fact was taken into consideration and it was tried to reduce the risk by getting the feedback from developers in many different positions and roles, so the survey will present a broad view.

5.2 Conclusion

To cope with the changing business requirements, the advancements of technology is a challenge which needs to be addressed by the business organizations in order sustain the position at the business industry. This has increased the importance of service based architecture for business software. The next level of service oriented architecture is microservices. This research provides with a methodology and an approach in migrating from a monolithic approach to service oriented approach through microservices. In the microservice architecture defined, one database server will be handled by a single microservice which means a change in one database will not affect the other services. This approach makes the architecture more flexible and indeed this provides more opportunity to scale.

1) The testing and the deployment can be automated.

If a simple modification or a major enhancement is done to a single microservice, this only requires the developers to focus on the source of the specific microservice since the testing could be automated. Also the each microservice is specialized for a specific business function and the interfaces between the microservices is clear.

2) Fault tolerant

Microservices are designed for fault tolerance. When one service fails it will be handled through an error handling mechanism and the other services could function without an issue. For example, when one of the microservices is not working, the system automatically routes the request to the healthy one. Even if there is no health microservice, the user can still add a condensed article.

3) Advantages of microservices derived from OData

The OData services derived provides both query and manipulation operations on data. OData provides an entire query language directly in the URL. This means that by changing the URL we

could change data set returned from the OData feed. This allows to filter through ‘sql-like’ syntax which uses keywords such as $top, $skip, $filter, $count, $orderby and operators such as ‘eq’, ‘contains’, ‘startswith’.

The $metadata end point is a part of the OData protocol. Some tools such as Visual Studio generates the entire proxy when the end point is simply defined.

In RESTful APIs, there can be some custom operations that contain complicated logic and can be frequently used. For that purpose, OData supports defining functions and actions to represent such operations. They are also resources themselves and can be bound to existing resources. Further, OData does not prevent joining data from multiple sources.

Some disadvantages of microservices over monolithic architecture was also identified during the research.

Complex configuration

When compared to the configurations in monolithic architecture, the configuration of microservice based architecture is not as simple as monolithic architecture. To achieve the expected results automation tools might be needed as monitoring tools, server environment, automatic deployment tools and automatic integration.

Resource usage is high

To achieve architectural flexibility in microservice based architecture, microservices use multiple tools such as Service Discovery and API Gateway. This requires more resources and this indeed increases the complexity of the system.

In order to face the challenge in deriving a service oriented architecture through micro services for a monolithic system, this research has provided a general approach for this migration. The proposed solution was to first extract the candidate services through Data Driven Design. According to the business domain and after analyzing the database schema, a set of candidate services will be derived from the monolithic system. Then the communication between the services needs to be considered also by considering the communication protocol, data format and microservices framework. If any selected candidate services seems inappropriate, those needs to

be rejected and when considering the communication between the services, if any two services has a higher communication overhead which could introduce a tight coupling between the services, those needs to be merged. The selected candidate services were designed as microservices using the OData protocol.

References

[1] E. Durou and P. Lee, "Deloitte's Predictions for the technology, media and telecommunications (TMT) sectors," 2017. [Online]. Available: https://www2.deloitte.com/content/dam/Deloitte/xe/Documents/technology-media-telecommunications/predicitons2017/ME-Predictions-2017-IT-as-a-Service.pdf. [Accessed 2018].

[2] Y. Zhao, "Service Oriented Infrastructure Framework," in *IEEE Congress on Services - Part I*, Honolulu, HI, USA, 2008.

[3] Columbus and Louis, "Roundup Of Cloud Computing," 2017. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2017/04/29/roundup-of-cloud-computing-forecasts-2017/#6487c81631e8. [Accessed 2018].

[4] M. Avram, "Advantages and challenges of adopting cloud computing from an," in *The 7th International Conference Interdisciplinarity in Engineering*, 2013.

[5] Y. Nosyk, "Migration of A Legacy Web Application To The Cloud," University of Applied Sciences, South-Eastern Finland, 2018.

[6] Almonaies, A. Asil and J. R. Cordy, "Legacy System Evolution towards," School of Computing, Queens University, Kingston, Ontario, Canada, 2010.

[7] M. Oliver, R. Florian and D. Schahram, "Domain-Specific Service Selection for Composite Services," in *IEEE Transactions on Software Engineering*, 2012.

[8] A. Grace, J. Lewis, E. Morris, B. Dennis and B. Smith, "SMART: Analyzing the Reuse Potential of Legacy Components in a Service-Oriented Architecture Environment.," May 2007. [Online]. Available: https://www.researchgate.net/publication/265739659_SMART_Analyzing_the_Reuse_Potential_of_Legacy_Components_in_a_Service-Oriented_Architecture_Environment. [Accessed June 2018].

[9] S. Yalezo and M. Thinyane, "Architecting and Constructing an SOA Bridge for an MVC Platform," in *Fourth World Congress on Software Engineering*, Hong Kong, China, 2013.

[10] C. Semith, A. Ilker, A. Oguztuzun and S. Tufekci, "A Mashup-Based Strategy for Migration to Service-Oriented Computing.," Middle East Technical University, 2007.

[11] D. Dagger, A. O'Connor, S. Lawless, E. Walsh and V. P. Wade, "Service-Oriented E-Learning Platforms: From Monolithic Systems to Flexible Services," IEEE, 2007.

[12] Z. Tari and J. Stokes, "Designing the reengineering service for the DOK federated database system," in *Proceedings 13th International Conference on Data Engineering*, Birmingham, UK, 1997.

[13] R. Terra and A. Levcovitz, "Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems," Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil, 2016.

[14] N. Kulkarni and V. Dwivedi, "The Role of Service Granularity in a Successful SOA Realization A Case Study," in *IEEE Congress on Services - Part I*, Honolulu, HI, USA, 2008.

[15] M. Razavian and P. Lago, "Understanding SOA Migration Using a Conceptual Framework," 2010.

[16] R. Maryam and L. Patricia, "Towards a Conceptual Framework for Legacy to SOA Migration," Department of Computer Science, VU University Amsterdam, Netherlands, 2009.

[17] Almonaies, A. Asil and Cordy, "Legacy System Evolution towards Service-Oriented Architecture," School of Computing, Queens University, Kingston, Ontario, Canada.

[18] S. Malik and D.-H. Kim, "A comparison of RESTful vs. SOAP web services in actuator networks," in *Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, Milan, Italy, 2017.

[19] M. Gavin and G. Denis, "A comparison of soap and rest implementations of a service based interaction independence middleware framework," in *Winter Simulation Conference*, 2009.

[20] K. S. Wagh and T. Ravindra, "A Comparative study of SOAP vs REST web services provisioning techniques for mobile host," *Journal of Information Engineering and Applications,* vol. 2, no. 5, 2012.

[21] K. Sangsanit, W. Kurutach and S. Phoomvuthisarn, "REST web service composition: A survey of automation and techniques," in *International Conference on Information Networking (ICOIN)*, Chiang Mai, Thailand, 2018.

[22] Y. Liu, Q. Wang, M. Zhuang and Y. Zhu, "Reengineering Legacy Systems with RESTful Web Service," in *32nd Annual IEEE International Computer Software and Applications Conference*, Turku, Finland, 2008.

[23] B. J. Kaviya and G. Selvakumar, "International Journal of Advanced Research in," November 2015. [Online]. Available:

http://ijarcsse.com/Before_August_2017/docs/papers/Volume_5/11_November2015/V5I11-0205.pdf. [Accessed 2018].

[24] M. Shang-Pin, H. Chun-Ying, F. Yong-Yi and K. Jong-Yih, "Configurable RESTful Service Mashup: A Process-Data-," *Applied Mathematics & Information Sciences,* vol. 9, pp. 637-644, Nov 2014.

[25] R. Chen, S. Li and Z. Li, "From Monolith to Microservices: A Dataflow-Driven Approach," in *24th Asia-Pacific Software Engineering Conference (APSEC)*, Nanjing, China, 2017.

[26] B. Mayer and R. Weinreich, "An Approach to Extract the Architecture of Microservice-Based Software Systems," in *IEEE Symposium on Service-Oriented System Engineering (SOSE)*, Bamberg, Germany, 2018.

[27] F. Wang and F. Fahmi, "Constructing a Service Software with Microservices," in *IEEE World Congress on Services (SERVICES)*, San Francisco, CA, USA, 2018.

[28] W. A. Brown, "IBM services blog," IBM, 17 September 2017. [Online]. Available: https://www.ibm.com/blogs/insights-on-business/gbs-strategy/can-microservices-next-enabler-innovation/. [Accessed July 2018].

[29] T. Huston, "SoapUI Pro," [Online]. Available: https://smartbear.com/learn/api-design/what-are-microservices/.

[30] F.-J. Wang and F. Fahmi, "Constructing a Service Software with Microservices," in *IEEE World Congress on Services (SERVICES)*, San Francisco, CA, USA, 2018.

[31] "using-microservices-for-legacy-system-modernization," 16 March 2017. [Online]. Available: https://www.altexsoft.com/blog/engineering/using-microservices-for-legacy-system-modernization/. [Accessed July 2018].

[32] P. D. Francesco, P. Lago and I. Malavolta, "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption," Universiteit Amsterdam, Netherlands, 2017.

[33] A. Joseph, "A simple overview on SAP Netweaver Gateway," 24 January 2013. [Online]. Available: https://blogs.sap.com/2013/01/24/a-simple-overview-on-sap-netweaver-gateway/. [Accessed June 2018].

[34] A. Kokkat, "Database Development with IBM Hybrid Data Architecture," 3 October 2017. [Online]. Available: https://www.ibm.com/developerworks/community/blogs/c4dd2a99-4d89-45b3-9931-8aff9e22b80b?lang=en. [Accessed August 2018].

[35] T. Clemson, "Testing Strategies in a Microservice Architecture," 2014. [Online]. Available: http://martinfowler.com/articles/microservice-testing/. [Accessed August 2018].

[36] M. Stine, "Migrating to cloud-native application architectures," 2015.

[37] T. Mauro, "Adopting Microservices at Netflix: Lessons for Architectural Design," 2015. [Online]. Available: https://www.nginx.com/blog/ microservices-at-netflix-architectural-best-practices/.. [Accessed June 2018].

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

[39] S. Newman, "Demystifying Conway's Law, 2014.," 2014. [Online]. Available: https://www. thoughtworks.com/insights/blog/demystifying-conways-law..

[40] P. Calcado, "Building Products at SoundCloud Part I: Dealing with the Monolith, 2014. building-products-at-soundcloud-part-1-dealing-with-the-monolith.," 2014. [Online]. Available: https://developers.soundcloud.com/blog/.

[41] C. Munns, "DevOps at Amazon: Microservices, 2 Pizza Teams, and 50 Million Deploys a Year," 2015. [Online]. Available: https://www.slideshare.net/TriNimbus/ chris-munns-devops-amazon-microservices-2-pizza-teams-50-million-deploys-a-year.. [Accessed 2018].

[42] W. L. Hürsch and C. V. Lopes, "Separation of concerns".

[43] M. Fowler and K. Beck, "Refactoring: improving the design of existing code. Addison-Wesley Professional," 1999.

[44] M. Richards, "Microservices Antipatterns and Pitfalls.," O'Reilly Media, Inc., 2016. [Online]. [Accessed 2018].

[45] E. Evans, "Domain-driven design: tackling complexity in the heart of software.," *Addison-Wesley Professional,* 2014.

[46] M. Fowler, "StranglerApplication," 2004. [Online]. Available: https://www.martinfowler. com/bliki/StranglerApplication.html..

[47] RaviShankarOjha, "Ecommerce with Shopping cart system," 31 July 2014. [Online]. [Accessed June 2018].

[48] Dennis, Enterprise Architecture Consultant and Microsoft vTSP., "Neuron ESB + Microservices + CQRS = Magic," 4 January 2017. [Online]. Available: https://www.neuronesb.com/article/neuron-esb-microservices-cqrs-magic/. [Accessed May 2018].

[49] "God Class," Cunningham & Cunningham, Inc, 2013. [Online]. Available: http://wiki.c2.com/?GodClass..

[50] Brown and Simon, "Modular Monolith," 2015. [Online]. Available: http://www.codingthearchitecture.com/presentations/ sa2015-modular-monoliths..

[51] G. T. Heineman and W. T. Councill, "Component-based software engineering.," *Addison-Westley,* p. 5.

[52] H. H. Ngoc, "Single Page Web Application with Restful API and AngularJS," Helsinki Metropolia University of Applied Sciences, 2014.