Tampere University

Bahareh Sadeghian Boroujeni

DEVELOPMENT OF A SHARED AUTHENTICATION SYSTEM – A MICROSERVICE APPROACH

Faculty of Information Technology and Communication Sciences
Master of Science Thesis
11/2019

ABSTRACT

Bahareh Sadeghian Boroujeni: Development of a shared authentication system – a microservice approach
Tampere University
Master of Science Thesis
11/2019
Major: Pervasive Computing
Examiners: Professor Kari Systä and Professor Davide Taibi

In the current market of business applications, Fatman Ltd. as a provider of several business services drew a conclusion that merging two related business services brings a competitive advantage to their market. Nevertheless, the integration of existing applications has some implications such as handling the cross-cutting concerns.

The authorization and authentication of the new system is a cross-cutting concern that needs to be resolved in the early stages of developing the application's integration. In addition, as a consequence of having a unified system a new user administrator tool is required to manage the users of different applications from a single place.

The purpose of this thesis is to find a solution for authorization and authentication by utilizing the microservices paradigm. Microservices is selected as an inspiration model due to the similarities of cross-cutting concern challenges in microservices and the new system.

The solution of giving access to users of different applications is developed as a single sign-on mechanism. This mechanism handles the user authentication of different systems and enables the user to have access to both applications through a single login.

In addition, to following up on the user authentication solution, the implementation of a new user management tool is demonstrated in the end. The new user management tool is designed to tackle the administrator users' problems which have been emerged by unifying the applications. The solution is made by following the domain-driven design and separation of concerns approach to simplify solving complex problems in monolithic applications.

The decisions on designing the new user access mechanism found to be well suited for the new system. The single sign-on mechanism serves the company for the upcoming development modernization plans as well as market enhancement. Moreover, the outcome of implementing the user management tool is evaluated by quantifying the number of service desk tickets regarding the lack of a proper user administrating tool.

Keywords: Microservices architecture, Single sign-on, Domain-driven design, Separation of concerns, Monolithic

PREFACE

This thesis has been done to carry out Master's degree studies in Information Technology with a major in Pervasive Computing. The thesis was written as an assignment for Fatman Ltd.

I would like to thank Fatman for providing this opportunity and support to conduct this research. I would like to thank prof. Kari Systä for all his guidance and support during the studies and writing my thesis.

I want to especially thank my family and friends who supported me during this work.

Helsinki, 12 November 2019

Bahareh Sadeghian Boroujeni

CONTENTS

LIST OF FIGURES

LIST OF PROGRAMS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application program interface
CM	Contract management
CRUD	Create, read, update, delete
CSS	Cascading style sheets
DB	Database
DDD	Domain-driven design
DI	Dependency injection
DTO	Data transfer object
ESB	Enterprise service bus
HTTP	Hypertext transfer protocol
IAM	Identity access management
IDP	Identity provider
JSON	Javascript object notation
JWT	JSON web token
Ltd	Limited
MRP	Maintenance resource planning
ORM	Object-relational mapper
RFC	Request for comments
SaaS	Software as a service
SAML	Security assertion markup
SoC	Separation of concerns
SSO	Single sign-on
XML	Extensible markup language

.

1. INTRODUCTION

Today's businesses are application-driven and organizations would not function sufficiently without software applications. As the Market Dynamics changes constantly, it is crucial to keep the applications up-to-date in order to witness expansion in business and gain an upper hand over the competitive market. In the era of ever-changing technology, innovative approaches in application management are rewarded as the applications must keep pace with the business strategy changes.

The enterprise application modifications often provide the capability of having a competitive advantage in the business market. For instance, providing a unique ability to a business application that enables the connection of disparate systems. This feature makes the business operation more efficient, eliminates manual work and accordingly, brings a competitive advantage to the market.

However, for the enterprise software vendors, the successful application modification to cater to the requirements of businesses while adhering to the high-quality standard of new technologies is challenging. Enterprise applications are complicated systems and the development of these applications requires delicate planning.

The enterprise applications have always been growing over the years in order to accommodate the new market tendencies. When the application gets too large it is hard to maintain and critical to make any functional changes to it. As a result, the idea of componentizing the large applications has emerged to face the challenges of maintenance and scalability in these systems [1].

1.1 Motivation

As a result of the consistent evolution of the market in business applications, Fatman Ltd. has decided to provide a competitive advantage to its software product. Hence, it has been decided to offer a unified system from separate applications with the matching look. These applications have related business use cases and integration of them enables Fatman to sell them as one product.

In order to fulfill this objective, the users of different applications should be able to have access to the whole system using a single credential. In addition, the administrator user

of the system should be able to manage the users of different applications from a single user management tool. Therefore, the implementation of a federated authentication for the integrated system with an associated user management tool is inevitable.

On the other hand, due to the size and complexity of the existing applications, the maintenance and scalability of the system became challenging. Accordingly, in this thesis, despite searching for a solution to the user management issues of the unified system, the overall improvement of the software is also considered. it is attempted to solve the user management issue while not making the current system more complex.

The user management issues in the unified system are similar to the challenges of adapting to the known architecture model, Microservices, and can be solved in the same way. The microservices approach can be pursued not only to solve the user management problems of the integrated system but also to be utilized for software modernization purposes.

Regarding the integration of existing applications, the thesis will strive to find solutions to the following questions:

- How to handle the user authentication and authorization of different applications in the integrated system by utilizing microservices?
- How to integrate the user management tool of the new system, considering the history of different applications?
- What are the best practices for the implementation of new features that support the software maintainability?

1.2 Structure of the thesis

In the theoretical background chapter, firstly the company products and portfolios will be introduced. Moreover, the different requirements of the system will be explained to provide a better understanding of the thesis purpose. Thereafter, the microservices architecture and its analogies to the new integrated system will be discussed.

Once the theoretical backgrounds are introduced, the third chapter will provide the methods and materials which will be carried out in the following chapters. In addition, the evaluation of methods, their advantages, and drawbacks will be introduced.

In the system design chapter, the applicability of the provided solutions in the integrated system will be discussed. Next, the chosen solutions and design by considering the given requirement of the new system will be demonstrated.

In the fifth chapter, the implementation of the user management tool will be provided in more detail. In the end, the resulting product will be evaluated to determine how well the new features serve the initial requirements and finally, the last chapter will summarise the thesis details and goals.

2. THEORETICAL BACKGROUND

The first section introduces the background information of the company and the product's portfolio. The second section specifies the software requirements which are provided as the business, functional and nonfunctional requirements. The third section gives instructions to the microservices model. In the 4th chapter, the database structure of Fatman products is provided to clarify the descriptions of the last section. In the last section, the similarities of microservices with the new integrated system as stated in the introduction will be specified.

2.1 Company and products portfolio

Fatman Ltd. provides enterprise applications to enhance the life cycle management of different businesses. The main products are maintenance resource planning (MRP) and contract management (CM). These applications are working on everyday devices, platforms, and browsers.

The MRP program, named Fatman frame, is designed to be used mainly by service companies, real estate owners, house managers, facility managers, and industrial companies [2].

The main features of the MRP application are composed of the basic information of real estates, plots and buildings; handling service requests of properties in a centralized system, forwarding service requests to the company suppliers, maintenance task scheduling, renovation planning, reporting and recording the meter data. The features are used based on different customer's business requirements.

Contract Management is another product provided by Fatman. It is a comprehensive tool for handling real estate and tenant information. Contract management's main functionalities are comprised of creating and sending invoices, fetching payment information from banks, managing ledgers, managing leasing contracts and creating bookkeeping material directly from the contracts. Moreover, this program is designed to work with MRP systems seamlessly.

In addition to the above-mentioned products Fatman also provides an application programming interface (API). This tool enables communication with other systems in case a customer has an own tool which requires data sharing with the Fatman frame.

2.2 System requirements

A large amount of business-critical systems that are vastly used by enterprises are legacy [3]. Fatman as a provider of several facility management systems is facing challenges of software modernization. The MRP system, Fatman framework, had been written in 2010. It has been converted from an old tool that became obsolete in early 2019. The other product, Contract Management has been in use for over 20 years. These applications have been growing over the years and the new functionalities were being added by the time of customer's need. As the applications grow, the maintenance and evolution of software applications become increasingly complex.

In this regard, despite specifying the business and functional requirements of the system it is advantageous to keep an eye on the overall improvement of the software as non-functional requirements. In this respect, the different requirements of the system are provided as follows.

2.2.1 Business requirements

The MRP application is mainly used by companies of real estate business. These companies are usually willing to have the contract management tool alongside. Currently, over half of the customers are using both applications and this number is increasing.

At this stage, the market of the MRP system has reached the point where Fatman is required to merge the contract management tool to the MRP system to offer the products as one. Therefore, Fatman demands an integrated system where two applications are merged with the matching look. By unifying the applications, the end-user is able to have access to both systems by a single login.

This unique feature in the market of business applications provides the opportunity of offering a comprehensive product that makes the business operations more efficient, eliminates manual work of end-user and enhances the user experience.

From a broader perspective, this approach will lead the company to attain a new future model for building and adding new tools. For instance, if it is managed to provide a solution that allows running multiple applications together as a whole, future features can either be an extension of the existing applications or alternatively an entirely new application.

2.2.2 Functional requirement

As explained in the business requirement, many customers of the Fatman products are using the MRP and CM applications alongside. Considering customer's demand for using two relevant applications side by side, some inconveniences have arisen to the users and administrators of the systems. For example, a single user has access to two different applications and is required to use both tools in order to accomplish a task. With the same credentials and roles, the same actions need to be taken repeatedly in order to authorize and navigate to the intended features.

The aforesaid issues bring some functional requirements for the users and administrator users of the system as bellow:

- The end-user of different tools should be able to have access to both systems as a whole.
- The user should have access to the comprehensive system by a single credential.
- The administrator user should be able to manage the users from a single user management tool.
- The user data should be consistent in all subsystems after the data modification.
- Navigation between the subsystems should be intuitive and easy to operate.

The above-mentioned requirements can be grouped into two categories. First, the user authentication in the unified system and second user data management.

2.2.3 Nonfunctional requirements

MRP and CM applications consist of several components. For example, considering the MRP system, it is made from different modules including handling of service requests, scheduling the maintenance tasks and planning renovations. Despite application comprises different modules, it is built and deployed in a single process, Figure 1.

In this single executable application, any changes to the system require the building and deploying a new version of it which is known as monolithic style. When the monolithic application grows, due to the size and complexity of the code it is hard to make any changes that ought to only affect a particular module. Accordingly, the scaling is required for the entire application rather than a single module. As a consequence, the maintenance and scaling of the application become challenging [4].

Correspondingly, in Fatman products, the maintenance of monolithic systems became challenging in particular when it comes to adding new features. Therefore, it is required to find a way out from enlarging the existing code whenever a new feature needs to be added.

In addition to the size and complexity of the code, the technologies used for developing the system become obsolete when the enterprise application becomes old. By sticking with the outdated technologies it is not feasible to improve the performance and user experience of the system. Hence, it is also required to figure out a solution by which the new features can be added using modern technologies. This approach would direct the development of products towards software modernization as a long-term investment.

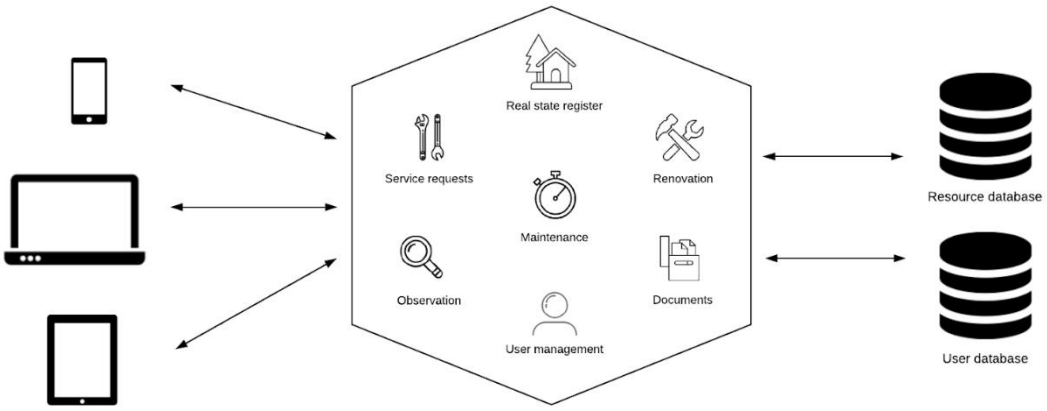

Figure 1. *A monolithic application consists of different components*

Considering the above-mentioned requirements, it is advantageous to study particular solutions which have made for similar architectural style. The next section introduces microservices architecture and its solutions to comparable issues.

2.3 Microservice architecture

Microservices architecture style describes an approach in software development in which designing of applications developed as a set of independently deployable services. The services are running on their own process and communicate with each other through a lightweight mechanism. [4]

The microservices approach has become a trend in the last decades due to the common reasons amongst many organizations. A large number of software companies possess big applications that have been growing in many years. They were building technical

debt over the years and at some stage, it is realized that the maintenance and scalability of the systems are getting highly complex. Consequently, they need an approach by which enhance the maintenance and scalability of their products. As a result, the ideas developed around splitting the applications up into smaller components running on their own process and talk to each other through the lightweight mechanism. In this way, the independently deployable services can be scaled and tested individually and thrown away if needed without affecting the whole system. The microservices can be written in different languages and own their individual datastore. This manner enables software companies to take advantage of continuous delivery and scaling applications in different axes. [1]

Figure 2. *Microservices with independent components*

One of the principles of microservices is to have a single responsibility for each service, Figure 2. A single responsibility can be defined as functional, non-functional or cross-functional responsibility. However, occasionally a single responsibility can become quite large. For instance, a payroll system in an online shopping application has a single responsibility but also not simple.

2.3.1 History of microservices

The “Microservices” term has been used extensively since March 2012 [5]. Some individuals from the microservice community claim that microservices demonstrate a new architecture style, while proponents from Service Oriented Architecture believe that microservices are an implementation approach to SOA.

SOA is defined in various contradictory ways by different groups. For instance, SOA for the community who have been into enterprise systems expresses an enterprise service bus (ESB). ESP is a vital component of SOA that is a pattern in which a centralized software component executes integrations in the background and make them available as a service interface to be reused by other applications [6]. On the other hand, Martin Fowler from microservices society defines SOA in a broad term and indicates that the microservices term can be a label as a subset of operation in SOA. However, some microservice advocates rejecting the label entirely.

Nevertheless, the microservice approach has been done by people under the name of SOA for at least a decade before emerging the microservice term widely. [7]

2.3.2 Some characteristics of microservices

There is no solid definition that can outline the microservices. However, there are common characteristics amongst the different microservice systems that are retrieved to describe the title. In the following sections, some common characteristics of microservices that are relevant to this thesis are selected. These characteristics are explained by Martin Fowler as following.

Componentization by services

Looking at the software industry over the years reveals that there has always been a desire to build systems by putting components together as in the physical world. Components are units of software that can be replaced or upgraded independently. In terms of software, there are two forms of components: services and libraries.

Libraries are pre-written codes that are obtained from third parties and linked to the program. Most of the programming languages have a collection of libraries which has been growing through the last decades considerably. Libraries are a part of the process and in order to call them, we are using in-memory functions. Replacing or upgrading the libraries does not demand a significant code changing.

Services, on the other hand, are different types of components that are running on their own process. Unlike the libraries, which are using in-memory functions, services use the

inter-process transmission such as web service calls. Services are more independent than libraries. For upgrading a library, the system should be compatible with the library version while services are independent components running in an independent process. Upgrading a service would not have any conflict with the other service components.[4]

Product development methodology

Studying the division model of large applications, such as enterprise systems, reveals that dividing the application into the parts commonly leads to splitting the software based on the technology layers. A common model is to form three different teams associated with each layer as front-end, back-end, and the data storage teams. In this model, even small changes in the code would lead to decision making across the teams. For example, a minor change in the server-side requires other team's approval.

Contrary, in microservices this type of management is fundamentally different. The services are built around the business capabilities and any individual team focuses on one service. The developers of these teams must fulfill the requirements around all the technology layers. Hence, every developer in the team is required to have full stack skills including the user experience, server-side and database management.

In monolithic applications, it is possible to divide the system into several modules based on business capabilities. In fact, the teams in developing monolithic applications are intended to split up themselves based on business boundaries. However, it is unlikely to end up with an optimal way of working in large systems. The main reason is when the business boundaries around the modules expand, all the members are imposed to remember the functionality of each business logic all over the system. This becomes problematic since it is not feasible for individuals to fit a large number of business-logic to their short memory. Thereby, more explicit separation of boundaries is required to clarify and enforce each team’s boundaries.[4]

Decentralized data management

In the monolithic architecture, a single database is used across the system in the interest of data persistency. However, in microservices decentralization is applied in many different ways as well as data storage. In this architecture, each service is allowed to have its own datastore as Figure 3. The approach is known as polyglot persistence. Polyglot persistence implies that it is permitted to have multiple data storage in one system which can also be implemented in different technologies. The data storage technology for each service is entirely up to the individual service. [8]

Having each service to be responsible for its own data may remove the complex integration through the databases. Moreover, it also enables us to choose a datastore technology for each service based on its compatibility with the corresponding component.

However, decentralizing the data storage across the system has some implications which will be discussed later.

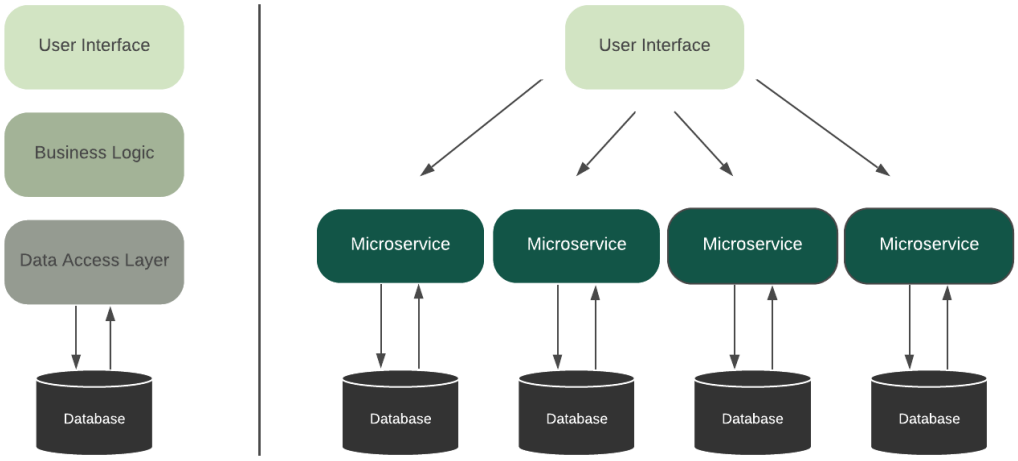

Figure 3. *Decentralized data storage in monolithic vs microservices*

2.4 Fatman database structure

Fatman Ltd. as a provider of the MRP system is following the Software as a Service (SaaS) delivery model. In the SaaS model, a single application is rented to multiple tenants as in Figure 4. In other words, each customer makes payments to the software company as a rent. In SaaS enterprise systems they provide customers a web-based software program with a more affordable pricing structure. As a result of the application being hosted by a service provider, the infrastructural costs are belonging to the vendors while the organizations only pay for the features they use [9].

There are different tenancy models in SaaS applications. Fatman products possess a pattern of multi-tenant applications with a database per tenant. In this pattern, a new database is made for each tenant (customer). The advantage of this model is the possibility of customization and optimization of databases per tenant since each customer has its own version of the database.

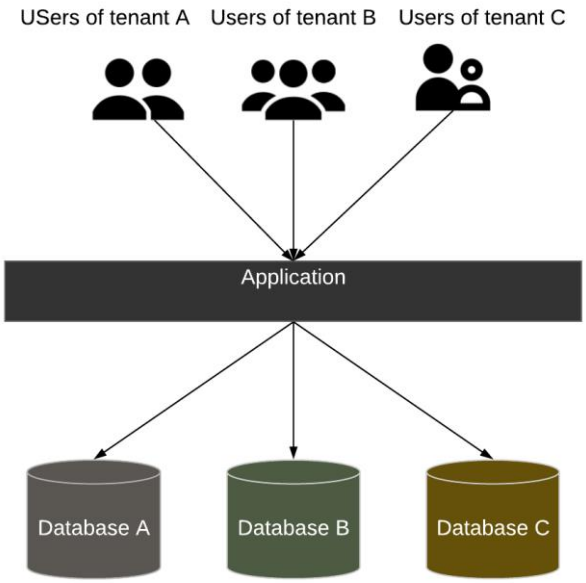

Figure 4. *Multi-tenant application*

Although each customer has its own version of the database, in Fatman products similar to the majority of enterprise systems, a single database is shared across multiple applications, Figure 5. The reason for that is the decisions made by commercial models of suppliers around licensing for this type of system. [10]

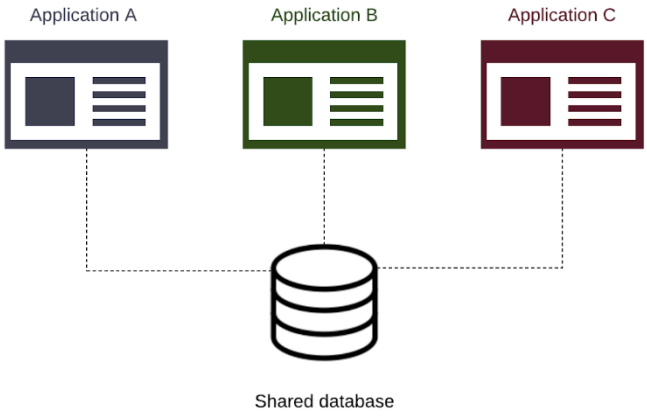

Figure 5. *Shared database for authorization and authentication*

While several applications are sharing a single database, separate databases containing the authorization and authentication information are in use by each application as in Figure 6. These databases are including the user's basic information, the company's information, roles, filters, permissions, messaging, etc.

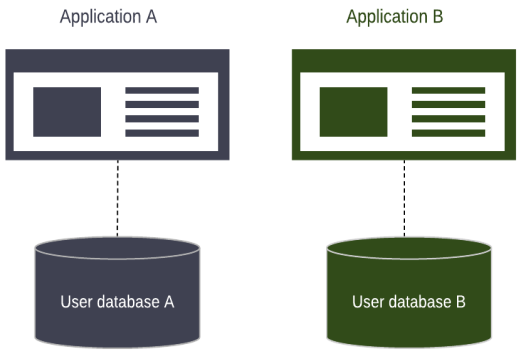

Figure 6. *Separate databases for authentication in different applications*

Hence the database structure, considering the users and products DBs resembles as Figure 7.

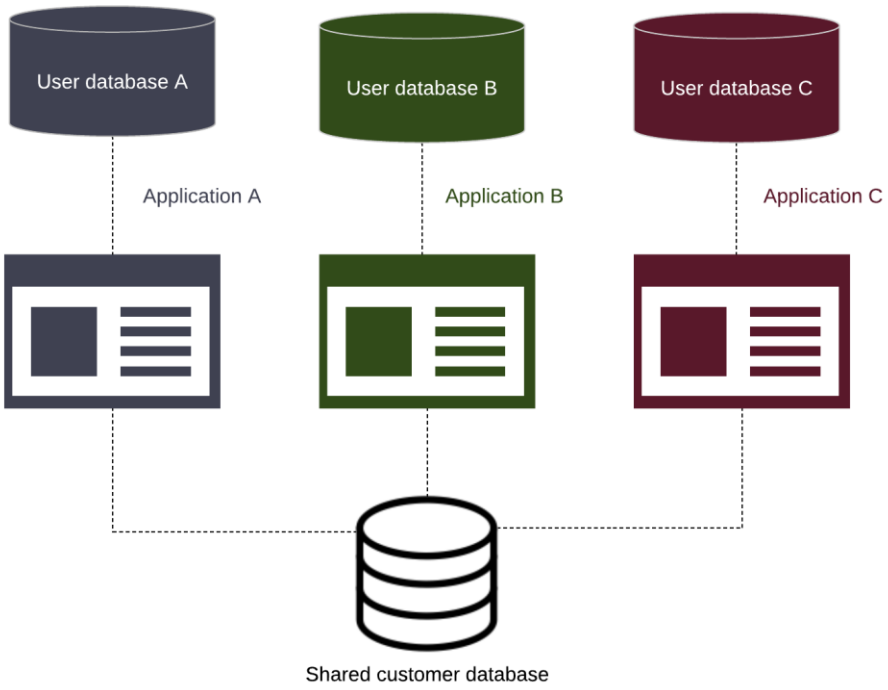

Figure 7. *Fatman products database architecture*

The user management tools of different applications only use the information from the corresponding authorization database. Considering the integration of two applications, a new user management tool needs to be developed by which it is possible to reach the information of both databases.

2.5 Microservices architecture analogies to Fatman integrated system architecture

In a microservices architecture, the system is componentized. Components are independent services that are designed to work together. As explained before, this software architecture model brings many benefits to the software system. On the other hand, at the same time, many problems of distributed systems, such as cross-cutting concerns arise. One of the main challenges is the production of a secure and efficient authorization and authentication mechanism.

Similarly, in Fatman, the unified system composed of different independent applications that are running on their own processes. Even though the applications are not small services, it has been decided to make them work together as a unified system. Thereby, the same challenges of authorization and authentication apply in this case.

Besides the authorization mechanism, there is another similarity between microservices and the integrated system. In microservices keeping the datastores consistent while the data is modified from various places is a concern. The concern results from the fact that each service has its own database and the same data can be stored in several databases.

As explained in the previous chapter, Fatman products possess different databases for authorization and authentication of each application. Imposing the applications to be a unified system demands the data consistency of user’s data which are stored in two different databases. The user's data is modified from various places; hence, it is needed to consider data consistency when moving to the new architecture.

In the following, these challenges are explained further.

Cross-cutting concerns

A system consists of several functionalities that are implemented by the main logics along with the tangled code. These functionalities are divided into primary and secondary based on the purpose they are made for. The secondary functionalities are related to some part of the code which is not associated with the main logic, for example, logging. Logging and the main code are from a different concern. A concern is determined as a

domain which is decomposed on the basis of the functionality. Concerns are usually found in different parts of the code but sometimes they overlap in one area. These concerns are known as cross cutting-concerns. Other examples of cross-cutting concerns are authorization, authentication, and configurations. [11]

Cross-cutting concerns are scattered across the whole system; hence a single function is repeated in several places. In monolithic applications, code duplication can be easily prevented. The solution is to locate the repeated code in a shared place that can be reused from various parts of the system. In contrast, in microservices, this cannot be an optimal solution due to the dependency that common code makes amongst the components. Therefore, cross-cutting concerns are problematic in microservices since it produces code duplication, chaotic architecture and complex maintainability.

There are several different primary cross-cutting concerns in microservices such as logging, exception handling, configuration management, monitoring, authorization, and authentication. In authorization and authentication as an example, each service should implement an identity authentication and authorization mechanism on its own due to the fact that services are independent processes. These functionalities are scattered throughout the entire system and make the architecture unorganized.

Hereinafter, the crosscutting concerns of microservices that are similar to the integrated system are explained more closely.

2.5.1 Authorization and authentication

In a monolithic application, when the user logs in through the security module of the system, a session ID is created for the user and sends it to the client. The client, which is the browser, restore this unique session ID associated with the user as a cookie. The session contains information about the user such as usernames, roles, and permissions. From this moment whenever a user sends a request to the server the session id is also sent to the application to verify the user's identity and permissions. Thus, the user doesn't need to enter the username and password for each request, Figure 8.

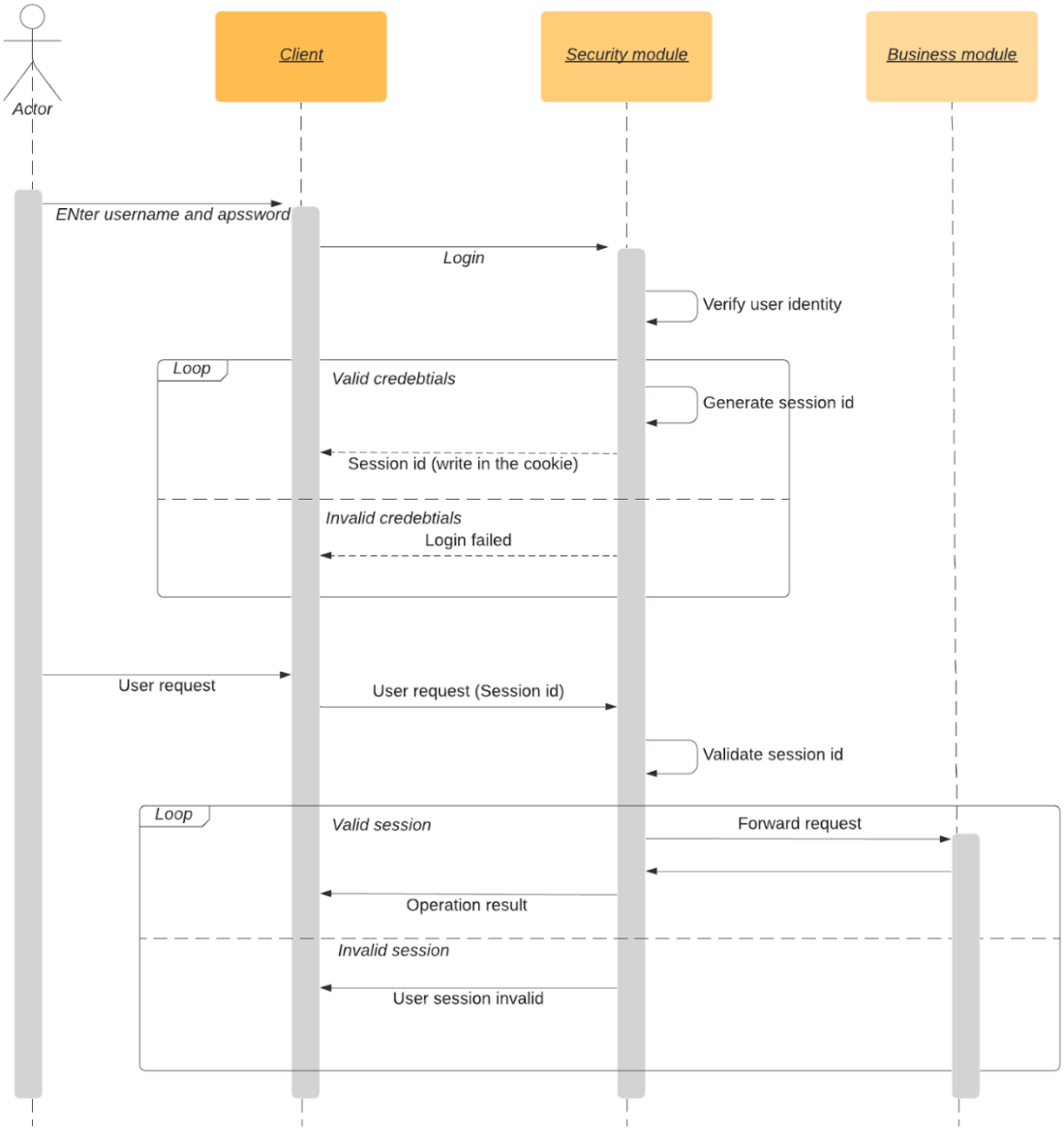

Figure 8. *Authorization and send a request in a monolithic system*

In microservices, a request to service can happen from an external or internal call. Depending on the design of software these requests can be authenticated either to a group of services or each service separately. If the authentication is made for a group of services, the internal calls do not necessarily require authentication.

Considering the similar situation az in monolithic systems, in microservices, each external request to the service needs to be authenticated and authorized due to the separation of processes in each service. For example, imagine a user who logs into a service that handles the service requests. This user also wants to create an invoice for the request which is handled in invoicing service. In this situation, the user needs to be authorized to the invoicing service along in order to accomplish the task.

Having a system with microservices architecture demands a mechanism for authorization and authentication which either gives access to the group of services or a single service depending on the external or internal service call types.

According to the above-mentioned issues, there are some challenges in the authorization and authentication of microservices:

- Authorization and authentication need to be handled in each service which causes duplication all over the code.
- Reusing a part of the code to avoid duplication causes dependency between the components.
- In microservices the rule of single responsibility per service must be followed, thus authorization and authentication responsibility needs to be handled in a separate component. [12]

2.5.2 Data management

In the microservices approach, decentralization applies to all aspects of software design. Decentralization not only guides the business logic organization of the system but also leads the data management. The decentralized data management is one of the basic principles of microservices which is presented in many ways.

The most abstract way of decentralizing data is to define a conceptual model for each component. This can occur when the application is split into separate components. A good example of approaching the system componentization is through the Domain-Driven Design of the bounded context (DDD). DDD divides up a complex domain into different bounded contexts and maps the relation amongst them. [10]

Apart from the conceptual models of data decentralization, in microservices, the data storage is also decentralized. Unlike the monolithic architecture that the whole system shares a single database, in microservices, each service manages its own data. The notion is known as polyglot persistence where multiple data storages with different technologies are used in a single application in order to take advantage of suitable techniques. [8]

Decentralizing repositories across microservices has some implications. The distributed approach to managing data causes duplication and data redundancy across the databases. Data redundancy leads to the problem of lacking data integrity and consistency. Unlike traditional data modeling, each entity can appear in several places and managing updates in this situation become troublesome. [13]

3. METHODS AND MATERIALS

This chapter explores the best practices and methods to follow up with the thesis objectives. The first section introduces the adopted research methodology to carry out the procedure in the thesis. The second section investigates the user authorization and authentication solutions to come up with the design in the next chapter.

3.1 Research methodology

This thesis strives to focus on the development and application of the task's knowledge in order to design artifacts rather than gathering theoretical information and solutions. Therefore, the "Design science" approach is utilized to accomplish outcome-driven research.

In the Design science methodology, unlike the Natural science, the main concern is about "devising artifacts to attain goals" [14]. Thus, the main concern is about functional performance improvement in the artifact. The design science products are evaluated against utility criteria. Hence, design science composed of two activities, build and evaluate. The building is the procedure of developing an artifact for a particular goal and the evaluation indicates the procedure of determining the wellness of the artifact performance [15].

The aim of this research is to design an artifact that will be developed and implemented and eventually evaluated against the requirements.

3.2 Authorization and authentication technical solutions

As described before, one of the challenges of adapting to the microservices is that without having a proper authentication and authorization mechanism, the global logic of authorization and authentication needs to be handled in each service. On the other hand, the principle of handling one business logic per service must be followed to keep the components simple. Accordingly, authorization and authentication functionalities should be centralized in one service. This service has one responsibility which is handling the user's authentication and authorization. Considering this situation, some solutions are provided as in the following.

3.2.1 Distributed session management

There are several options to maintain the user login status after the user is authenticated. One way is to save the user data on the server as a session. In this way, when moving between the services, each request from the server remembers the user's identity status. There are several options to implement this procedure:

- **The sticky session**, by which, it is ensured that all the requests from a user are handled through the server which has handled the first request. Accordingly, whenever a request is sent to the server from any service, the server remembers the user's identity [16]. However, this solution may fail by the load balancer. If the load balancer decides to force the user to switch to another server, the user's session will be lost, and the user will be thrown out of the system.
- **Session replication**, which is a mechanism used to replicate the session between different instances of servers which are part of the same cluster. In this mechanism when the session data is changed, it is needed to go through all the instances for synchronization. [17] The synchronization may cause bandwidth shortage. The greater number of instances makes more overhead to the bandwidth.
- **Centralized session storage** in a sense of having a shared place where the session is stored and each service will have access to the session. The downside of this solution is the lack of security. In this solution, special protection to the shared session is needed. [12]

Despite the above-mentioned drawbacks, in all the stated solutions the server must be stateful. Having a stateful server impacts the horizontal scaling of the system. Moreover, for security reasons, there is the concept of the "same-origin policy". The same-origin policy is enforced by the browser to dictate the cookies to be accessed only by its creator. In other words, when there are different applications and each one has a different domain, the cookies of one domain cannot be accessed from another domain [18]. For this reason, another solution to share the session information across the domains needs to be identified.

3.2.2 Client token

Another option to share the login information is to generate **Tokens** to record the user's login status. The difference between tokens and the sessions is the place where they are stored. The sessions are kept by the server while the tokens are remained by the user. In this way, the server does not need to be stateful. The tokens are either stored

directly in the web store or in the cookies. Considering the tokens contain the user's credentials and carry sensitive information, they have to be treated cautiously. [19]

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. The JWT tokens are digitally signed and encrypted to ensure that information is verified and trusted. [20]

In order to share the authentication information across the different domains, a central domain is needed to perform the authentication. When the user is authenticated, a signed JWT token is generated in the central domain which will be shared with the other domains. This token contains the required information to identify the user in any other domain. Considering the token information is encrypted, the content cannot tamper along the way.

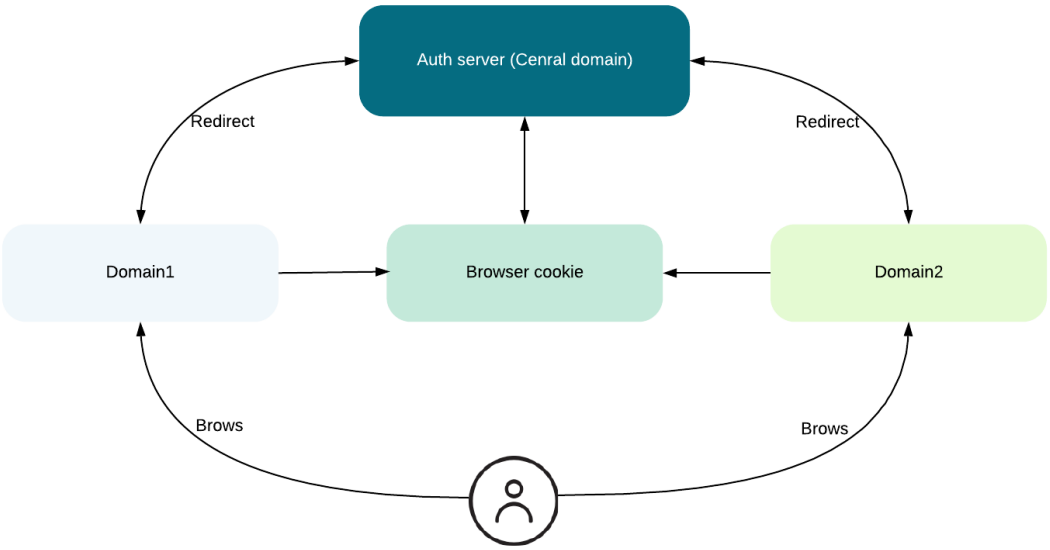

Figure 9. *Central authentication domain mechanism* [18]

As demonstrated in Figure 9, whenever the user goes to a domain for authentication, he will be redirected to the authentication domain. The user inserts the credentials and as soon as he is authenticated, he will be redirected to the original domain. From this moment the token is generated, and the user has access to the other domains as well. As a result, if the user wants to use another domain, the authentication will be verified by the authentication domain and the user is redirected to the second domain.

As described earlier, for the purpose of taking full advantage of microservices features, it is beneficial to distinct the user login responsibility in a separate service. In order to

address this objective, the Federated identity management approach is introduced as bellow.

3.2.3 Federated identity management, SSO and technical solutions

Federated identity management is defined as a system that enables the user's identification data to be usable across distinct identity management systems. In this manner, each identity management system is able to acquire access to the network of all applications in the group. [21]

Accordingly, from the end-user point of view, federated identity management establishes single sign-on access to all the applications across heterogeneous domains. Meanwhile, there is no need for the development of homogeneous databases for identity information. Each application maintains its own data store to be used for authorization and authentication.

Federated identity system handles several concerns namely, authentication, authorization and user management. Authentication refers to validating the user's credentials and generating the identity of the user. Authorization indicates the user's access restrictions and user management is related to the administration of the user accounts.

Single sign-on (SSO) refers to the authentication part of the federated identity system and has a simple context. It implies that whenever the user has authenticated to the application with a domain A, he also has access to the other application in domain B. This can be applied in a network of applications meaning that the access can be given to all the subsystems of a comprehensive system. This task is done based on the mechanism of each user-oriented request should go through the authentication service. If SSO approves the request, it grants a session to the user to access the intended application. This session is generated in a form of aforementioned JWT token. [18]

Single sign-on is a common solution to authentication and authorization in microservices. The federated identity system functionality is encapsulated in a microservice to avoid duplication and dependency between the services. Thereby, any changes to the service do not affect the other components of the system.

There are many ways to implement the SSO mechanism, in the following the more details are explained.

SSO technical solutions

With the aim of having a federated identity, it is needed to decide about an authentication protocol to transfer the authentication data. There are different authentication protocol

options that are used in federated identity. As of today, there are two protocols that are most adopted and popular in modern web applications, OpenID and SAML. [22]

SAML Security Assertion Markup Language is an XML based set of standards that are designed to be used in SSO. It provides a data format to exchange information about authorization and authentication between the identity provider and the service provider. [23] SAML is a product of the OASIS security services technical committee from 2001 [24].

OpenId Connect is an identity protocol that allows users to be authenticated using identity providers. It defines an ID token for returning the user information. It is an interoperable secure identity layer on top of OAuth 2.0 specification which allows the authentication session information to be accessible across several applications. [22]

OAuth2 is a generic access authorization protocol that enables applications to get limited access to the user's accounts on HTTP services. OpenId Connect is an OAuth extension for defining the id token. Thus, OAuth2 grants authorization while OpenId connect defines an ID Token for authentication.[25][26]

There are some benefits to choose OpenID in preference of SAML.

- OpenID easily uses identity token and the client receives this token in a JWT format, known as id token. The benefit of having the ID token encoded by JTW is the wide range of signature and encryption algorithms it claims.
- OpenID uses the OAuth2 protocol which has its own benefits. OAuth2 is used by clients to get the ID token. It means that a single protocol can be used for authentication and authorization.
- OpenID is yet simple to use despite its capability.
- OpenID is simple to integrate with basic applications while offering security which is crucial for enterprise systems.
- OpenID has security options that can meet enterprise requirements. [27]

A typical SSO mechanism is illustrated in Figure 10 as below.

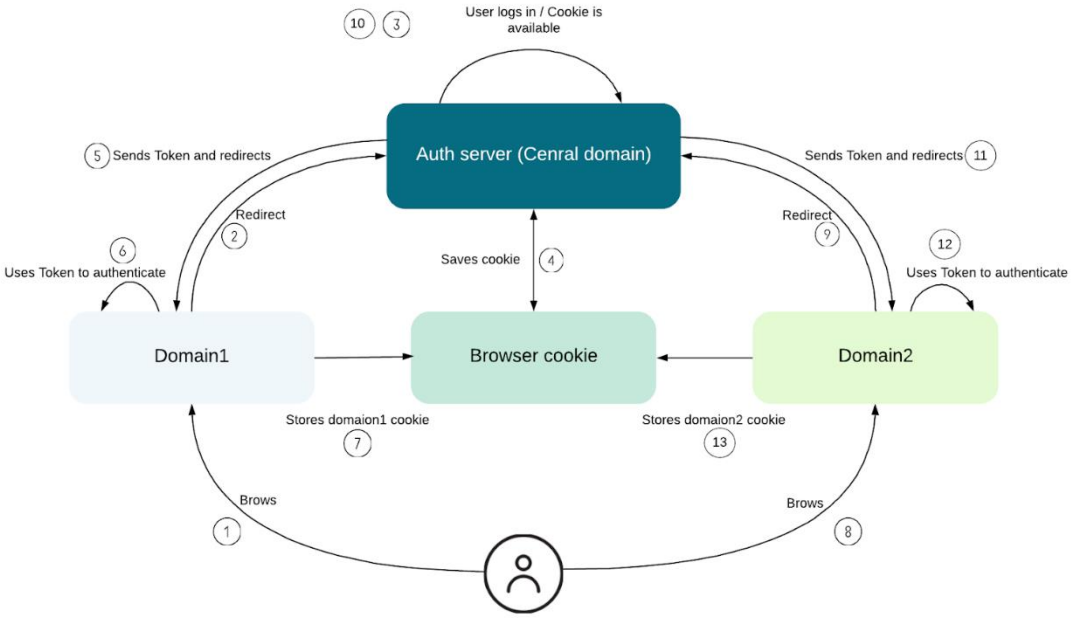

Figure 10. *A typical SSO mechanism* [18]

4. SYSTEM DESIGN

In the theoretical background chapter, the problems of authorization and authentication in microservices and its similarities to the Fatman framework are described. In the previous chapter, some solutions to the authorization and authentication of the distributed systems are introduced. The following sections evaluate the applicability of the provided solutions in the Fatman framework and provide the architectural decisions with respect to the microservice's paradigm.

As previously stated, it has been decided to integrate the MRP and CM applications. Due to the size and complexity of the existing monolithic applications in addition to the vital business processes in the customer organizations, refactoring a large amount of code is not an optimal solution for the accomplishment of this task. It is rather more beneficial to revamp the existing code by inspiring from the Microservices architecture to make the existing applications work together.

Making the existing tools work together to provide a holistic experience while avoiding refactoring, enables the development of a new architecture model. In this model, a new tool can be added without interfering with the existing tools. Therefore, it is not necessary to lock with the old technologies, the new tools or features can be added to the comprehensive system whenever needed with a desirable technology stack.

As previously stated, the Fatman integrated application consists of two independent subsystems with identical identity management mechanisms. Each identity management system uses a separate database for handling the authorization and authentication.

Accordingly, each application has a separate user management tool to be used by the administrator of each system. Considering this situation and the aiming of integrating two applications, firstly a new mechanism for authorization and authentication and secondly a new implementation of user management tool is provided.

4.1 Authorization and authentication

The MRP and CM products are created to serve different businesses but also they have some related business use cases. For example, consider a user from a maintenance company who has access to both MRP and invoicing applications. Due to the associated businesses, the user is required to use them together in order to complete the desired task.

In this use case, the user wants to first use the MRP system to create a new service request. Afterward, an invoice needs to be created for the customer who the service is requested for. In order to perform the task, there are several actions to be taken:

- Authenticating to the MRP system
- Making a new service request
- Logging out
- Authenticating to the invoicing system
- Making the invoice
- Logging out

The similar steps need to be taken by the maintenance personnel in order to follow up on the service request's tasks as illustrated in Figure 11.

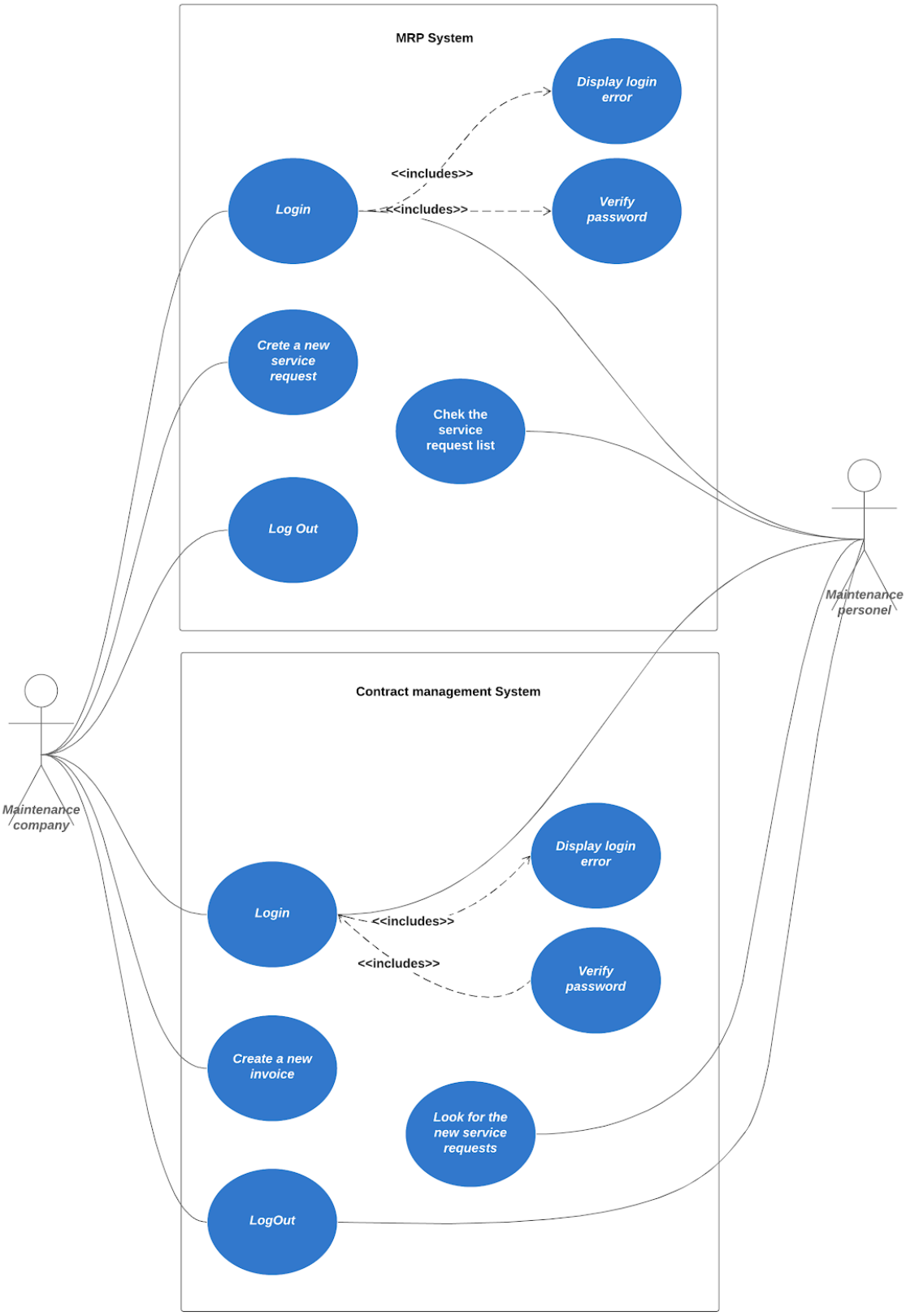

Figure 11. *Use case diagram of using different business applications side by side*

As can be seen from the diagram, some actions, like the process of authentication, can be more complex when the user has forgotten the passwords or the credentials are inserted incorrectly. Another example is when the user realizes some basic information in the service request needs to be modified after invoice creation. Evidently, the whole system can be improved due to the following reasons:

- Avoiding repeated actions for authentication, navigation and browsing the different domains.
- Eliminating the credential validation for several times.
- Preventing password fatigue which entitled to the experience of remembering several passwords.
- Reducing the probability of user error while switching between applications.
- Avoiding poor user experience and loss of time.

In order to eliminate the troublesome actions mentioned above, we need to find a way to share the user's login status to the other application whenever the user is authenticated from any one of them. Hence, we need a new authorization and authentication mechanism. With respect to the componentization principle from microservices, this mechanism should be encapsulated in a separate component to handle only the responsibility of user authentication and authorization. In this way, we will have a separate process for handling user authentication which is easier to scale and maintain, Figure 12.

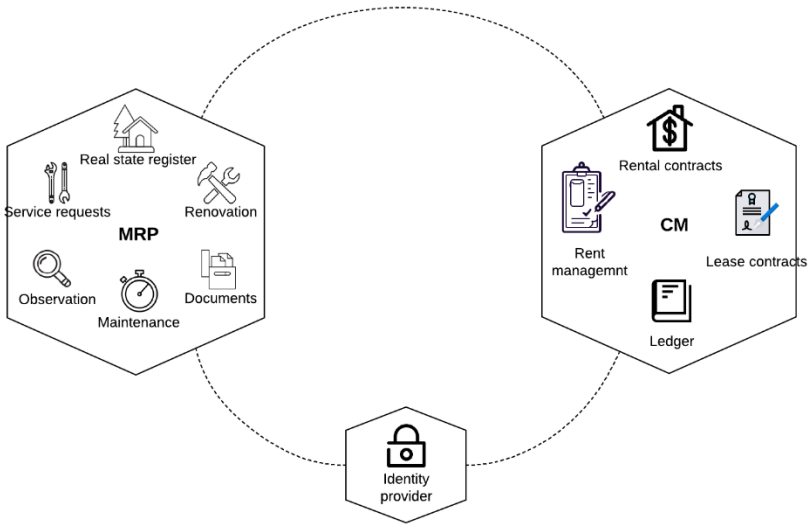

Figure 12. *Separation of identity provider component in the new integrated system*

In this case, it is required to share the login information between the applications of different domains. In the previous section, the different solutions to a similar issue in microservices are explained. Likewise, in Fatman products, a central domain should be implemented in order to generate a JWT token to share the session information amongst the subsystems safely.

In this way, when the user logs in to any service provider, instead of providing the credentials to each application, the credentials are provided to the Identity Provider (IDP). The IDP is trusted by the applications and the user's credentials are validated by IDP. Hence, the IDP federates the service providers and deliver an SSO mechanism.

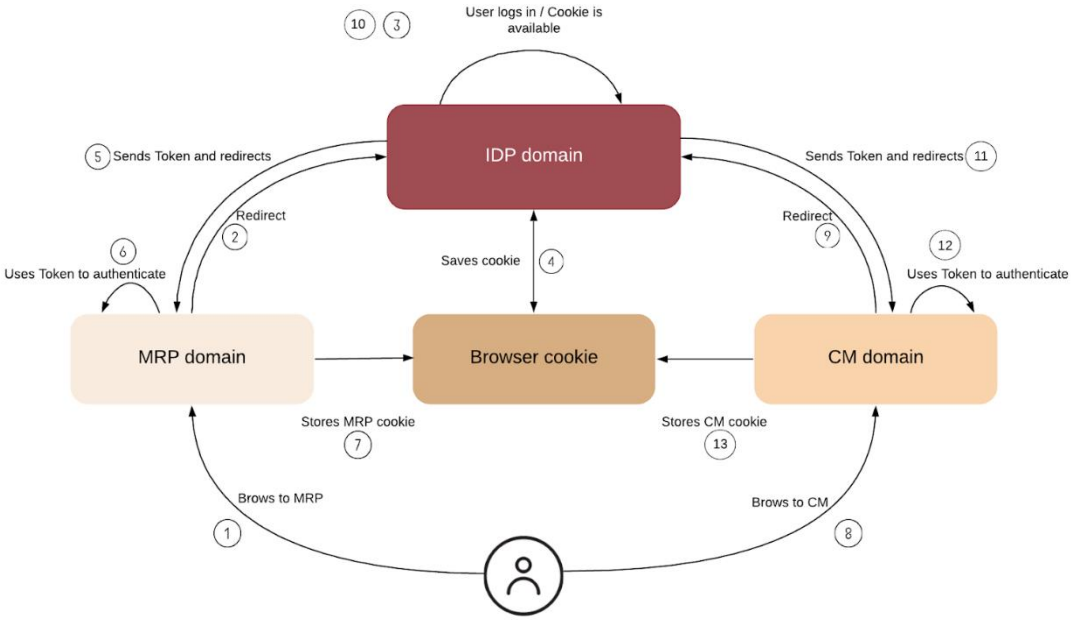

Figure 13. *Single Sign-on in Fatman products*

As can be seen from Figure 13, when the user browses the MRP.com, he will be redirected to the shared domain which is IDP.com. The user needs to enter the credentials in the shared domain. The credentials are validated by IDP and a token is generated. Afterward, he will be redirected to MRP.com to be authenticated with the JWT token. Later, when the user navigates to CM.com, he would have access to the application already. The same process happens if the user first browses to CM.com or in the future from any other domain. [28]

Having SSO in the Fatman unified system ease the process of adding new applications to the distributed system without worrying about internal authentication. Implementing the IDP in a separate service provides the possibility of independent development and deployment without interfering with the complex monolithic applications. This solution is a proper fit in Fatman since it can also ease the future plans of software modernization.

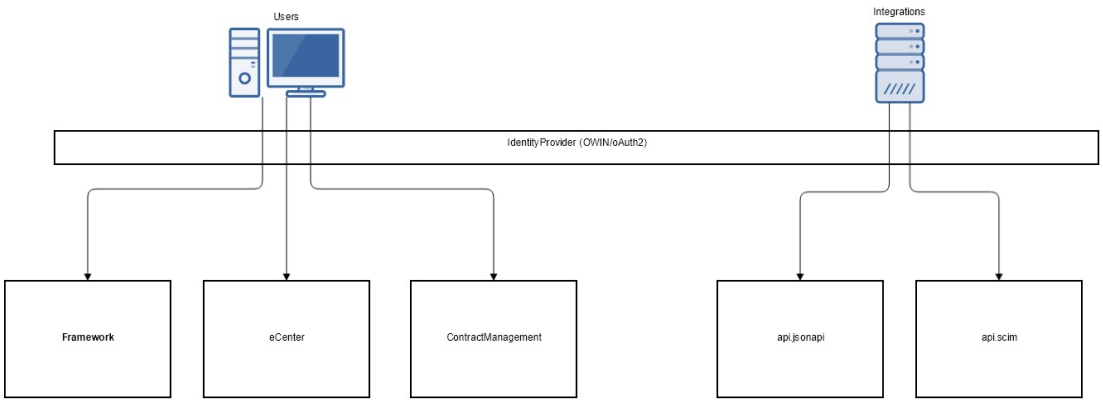

Figure 14. *Fatman products using IDP for authorization and authentication*

In order to follow the solution of having federated identity management, there are three main functionalities to be implemented:

- User mapping: Considering the existing users who already have access rights to either of the applications. There should be a mechanism to map the existing accounts of different applications to be used in the federated identity provider. Therefore, by signing in to the identity provider, they can automatically sign in to the relevant applications.
- User registration: The new user accounts should have the default access to both systems. In other words, the user must be able to use this account to have access to both systems. The user's authority to the different modules of the application should be handled in the authorization.
- User authentication: This function provides the user identity authentication. When the user signs in to the identity provider, an access token is created to be shared with the applications afterward.

The user authentication process is described as in microservices and the solution to the user's mapping and registration will be discussed in the following sections. Later, the implementation of user mapping and registration will be introduced in more detail in the next chapter.

4.2 User data management

By implementing the federated identity management and unifying the systems some implications come into begin. The unified system is encountered as a single product and the user information databases are decentralized. Undercover, there are different schemas and configurations for each system. The configuration settings and user management tools are scattered and modifying a user through one of the tools, results inconsistency between databases.

In the new system, the administrator user should be able to add or remove users as well as modifying their information and give them access and rights to different parts of the system. These requirements could be fulfilled simply when they only have the MRP system without Contract management addition.

Nevertheless, In the new system, the user modification operations only affect the user in one system and remain the equivalent user in the other application intact due to the decentralized data stores.

In this situation, there are several problems arisen as below:

- Overload of work for the service desks of the company associated with setting up and modifying a user in different databases.
- Using several tools for setting up an identical user.
- Twice the effort for any conventional user management operations.
- Redundant data due to the distributed data stores.
- Using manual scripts constantly for maintaining data consistency between the applications.
- Overload of work for development on error occurrence due to data inconsistency.

Consequently, user management in this system is time-consuming, error-prone and lack of access control causes difficulties in the future.

An entirely fresh design of the user administrator tool is not ideal in the current situation because of the following reasons. Fatman has several administrator tools, the company-level admin tool, as well as the user management tool which is used by customers. These tools are essential for everyday usage of project managers, service desks and system administrators. Malfunctioning of these tools causes critical problems to the end-users such as preventing users to access the applications. Designing a fresh application to be a separate component requires a lot of resources and is time-consuming.

Considering this situation, it is more beneficial to implement a mechanism for managing users with new technologies while being able to reuse the existing code. Reusing the existing code and minimizing the load of refactoring, speeds up the process of user management implementation with less amount of resource consumption. Moreover, it will be implemented in a certain way that does not have confliction with the future plans of software modularization and loosely coupling.

According to the studies, in microservices architecture, a successful design is made by modern advanced web application development and recent software engineering paradigms in particular Domain-driven design [29]. DDD is a model-based development approach that is guided by rules such as bounded organization contexts and continues software integration [30].

Domain-driven design solves the problems of applications by breaking down the knowledge and turning the chaotic information into a practical model. A model-driven design connects the model and the implementation closely by using Ubiquitous Language which is a common, rigorous language between developers and users. Thereby, the result is software that provides robust functionality based on a basic understanding of the core domain. [31]

DDD acknowledges that "total unification of the domain model for a large system will not be feasible" [31]. For this reason, DDD divides up a large system into Bounded Contexts, each of which can have a unified model. Bounded Context is a core pattern in Domain-Driven Design. In fact, the focus of DDD's strategic design to deal with large models and teams is bounded context. DDD divides the large models into different Bounded Contexts and makes the interrelationship explicit. [32]

In microservices, as well as decentralized decisions about abstract models, the data storage decisions are also decentralized. [10] One of the main structural patterns encountered in DDD is the Repository Pattern. When a persistent domain model is created, it is needed to retrieve the objects from an encapsulated data store. Repository Pattern introduced as a part of DDD in Eric Evan’s book when it published in 2004 [31].

Repository pattern

Repositories pattern introduced as a part of Domain-Driven Design which provides an abstraction to the logic of accessing the data. In the complex domains with a large number of classes and heavy querying, it is worthwhile to have a layer of abstraction to centralize data access and present better maintainability. It also encapsulates the technology used to access the data from the domain layer and aids with loose coupling in addition to minimizing the duplication of queries. [33]

As Martin Fowler describes: “A repository performs the tasks of an intermediary between the domain model layers and data mapping, acting in a similar way to a set of domain objects in memory” [34]. On the client-side, the objects make queries declaratively and send them to repositories to fetch the data. The repositories encapsulate the data stored in a database and their operations in a set of objects. The repositories support the separation of the work domain and data mapping as illustrated in Figure 15. [35]

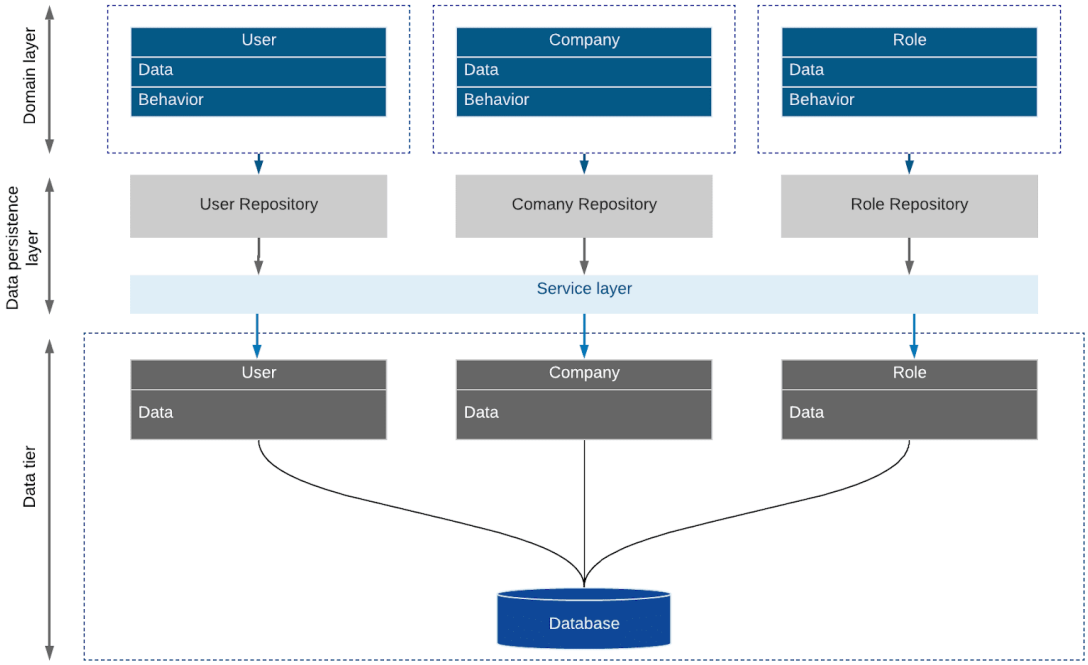

Figure 15. *The relationship between repositories, aggregates, and database tables* [35]

Advantages:

- Repositories make the unit testing easier. Having the repositories as an abstraction of the infrastructure makes it possible to test the infrastructure which is not possible in the unit test.
- In the legacy data access pattern, the data access object directly accesses data storage and makes the operations against the storage. In the repository, it marks the data with the operations in the memory of the unit of work and doesn’t perform the operations immediately.
- Using a repository pattern with a unit of work leads to better performance and consistency. The unit of work refers to a single transaction that contains multiple operations of Create, Read, Update and Delete (CRUD). It runs the operations in a single transaction. [35]

As indicated earlier, there are several tools and features that apply modifications to the user information. In particular, the user management tool and user profile in the MRP system and the company-level user administrator tool. Consequently, there are duplicated functions for the same operations all over the code. In this situation, it is beneficial to centralize these operations in a commonplace that can be used from different controllers.

In the repository's pattern, it is common to implement a higher Service layer as demonstrated in Figure 16 to interact between the controllers and the data access layer. Implementing a service layer aids, to have a single place for business logic and avoid duplication in the lower layers. The service layer which uses the repositories is middleware where we can implement the required functions for user management and call the repositories to access the correct database for CRUD operations. This service layer is in a shared library and independent of any application.

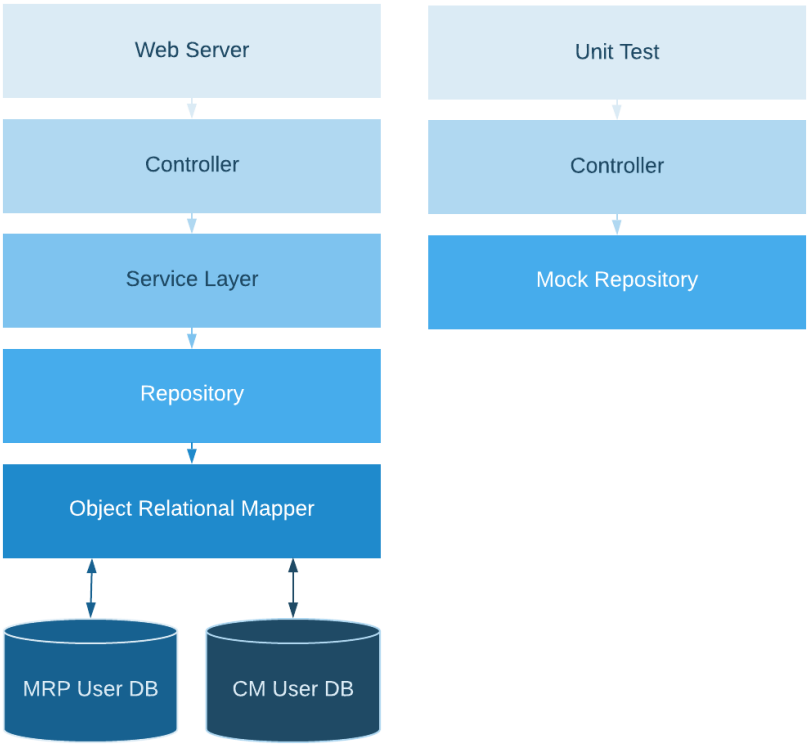

Figure 16. *Using repositories as an abstraction layer between database context and controller with a unit of work possibility*

The implementation of repositories and the service layer from the diagram in Figure 16, is not dependent on any application and can be used from several places. In this regard, it does not prevent the decomposition of the user management tool in the future. In the next chapter, the implementation of the illustrated diagram is presented in detail.

5. IMPLEMENTATION OF USER MANAGEMENT TOOL

This chapter describes the development of the user management tool with the adopted decisions and designs introduced in the previous chapter. The tool is developed to meet some part of functional requirements by considering the software modernization approach which will be pursued further in the company's future plans.

The project is implemented in C# language using ASP.NET framework in server and client-side applications. HTML, CSS and JavaScript are utilized in front-end development. The process of tracking the changes of source code is done by Git version control. SQL Server Management Studio is used for managing databases and building scripts for primitive data mapping.

The development consists of building the user mapping script and implementation of repositories. It continues to the development of business logic layers and in the end, unit testing development is introduced in order to validate each unit’s design.

The architecture decisions for implementing the user management tool in the following sections are given by the senior developers with respect to the current system architecture and aiming of using new technologies. The implementation of the new tool is the thesis work which is demonstrated as follows.

5.1 Data mapping

In order to implement the user management tool in the unified system, primarily it is needed to provide a list of all users of the customer in the main user management view. The users belonging to each company are classified into three different groups. First, the users who only have access to the MRP system, second, the users who only have access to the CM system and third the users who have access to both applications. Considering the third group, users who are included in both databases need to be mapped and displayed as a single user.

In order to gather all the above-mentioned users, there are two proposal methods:

- Fetching the list of users from each database individually in the code, mapping the third user groups and finally create a list of user’s objects and apply the operations.

- Mapping the users of different databases by building a script and thereafter inserting them into a new table. The list of all the users from different applications can be fetched directly from a single table. This table would be used as a link to the original tables in order to apply operations.

Creating a new table assists us with risk mitigations as below:

- In an identity provider, it can be verified that both users of different applications are using the same database in case the table has corrupted data.
- The data could be a sanity check in the table by joining both the user's structure and comparing the connection strings.

Because of the mentioned reasons, it is decided to build a script in which the users of different databases are mapped and inserted to a new table. The mapping is done through the connection strings and the unique login value of users. The new table will be in use for Identity Access Management (IAM). The simplified user mapping script is provided in appendix A.

5.2 Repositories implementation

As explained in the previous chapter, the repository pattern is selected as an infrastructure persistence layer in the new tool. In addition to the infrastructure persistence, repositories support the separation of dependencies within the work domain and the data mapping. That is to say, with the implementation of the repository layer, the data layer has a clean separation from the business layer. By splitting the responsibilities of each layer, a clean example of Separation of Concerns (SoC) is achieved [35].

To start with repositories implementation, it is necessary to define one repository per aggregate. The aggregate or aggregate root is the entity that behaves like a parent for other sets of related entities. In the domain-driven design, the only way to communicate with the database is through the repository since an aggregate has a one to one relationship with the repository. In this way, the repository controls the aggregate transactional consistency. [35]

In the MRP authorization database, there are four specified aggregate roots as users, companies, roles, and filters. In CM authorization database these aggregates exist as users, groups, and companies.

Repositories are written as classes that encapsulate the data access logic. Given that each repository is implemented per aggregate, the type of the entity that a particular

repository is working with should be enforced. As a result, the repositories are implemented with the interface classes. Each repository has its own interface and communicates with the business layer through the corresponding interface. The user aggregate interface is written as Program 1:

```
namespace MRP.Repository.Authorization
{
    public interface IUserRepository
    {
        //...
    }
}
```

Program 1. *User aggregate interface*

And the user class to implement the user interface is written as following Program 2:

```
namespace MRP.Repository.SqlServer.Authorization
{
    public class UserRepository: IUserRepository
    {
        //...
    }
}
```

Program 2. *User repository class*

After implementing repositories with their interfaces, we create the models to represent the domain-specific data. The model in repositories should encapsulate the data to transfer them between the subsystems. Therefore, we are using the Data Transfer Objects (DTO) in models to represent the data in the domain as in Program 3. In this manner, the persistence model is decoupled from the service layer model. In other terms, the DTOs are not containing any business logic, rather they transfer data through the entity bean and the higher service layer [36].

```
namespace MRP.Repository.Authorization
{
    public interface IUserRepository
    {
        //...
    }

    public partial class UserDto
    {
        public int ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Login { get; set; }
        public string Email { get; set; }
        //...
    }
}
```

Program 3. *Implementation of user DTO*

Repositories considered as a set of domain objects in memory. Consequently, any business logic implementation in the repository layer should be avoided. For this reason, only a set of generic functionalities are allowed to be implemented which perform the CRUD operations [35]. The example methods are defined in the interface as Program4:

```
namespace MRP.Repository.Authorization
{
    public partial interface IUserRepository
    {
        UserDto[] GetUsersById(int Id);
        UserDto[] GetAllUsersForCompany(int MRPCompanyId);
        int DeleteUser(int Id);
        //...
    }

public partial class UserDto
    {
        public int ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Login { get; set; }
        public string Email { get; set; }
        //...
    }

}
```

Program 4. *Method's declarations in the user interface aggregate*

In order to query and manipulate data from a database and using an object-oriented paradigm, we are using Object Relational Mapper (ORM). ORM provides maintainability and loosely coupling as a layer between a software application and the database. The ORM maps database tables to the C# entities and abstracts DB systems. There are

different libraries that can serve this purpose. Dapper is a simple object mapper with an optimal performance amongst the other ORM libraries. Dapper maps .NET objects from the application to a relational database and vice versa. [37]

Thereby Dapper is chosen as an ORM layer between the applications and databases. Here is the implementation of methods in a repository class using Dapper in Program 5:

```
namespace MRP.Repository.SqlServer.Authorization
{
    public class UserRepository: IUserRepository
    {
        public UserDto[] GetUsersById(int Id)
        {
            using (var connection = GetDbConnection())
            {
                return connection.Query<UserDto>($@"select {AllSqlColumns}
                                                    from [User]
                                                    where UserID=@Id",
                            new { UserID})?.ToArray() ?? new UserDto[] { };
            }
        }

        public UserDto[] GetAllUsersForCompany(int MRPCompanyId)
        {
            using (var connection = GetDbConnection())
            {
                return connection.Query<int>($@"select {AllSqlColumns}
                                                    from [User]
                                                    where CompanyID=@MRPCompa-
nyId",
                            new { MRPCompanyId}).ToArray();
            }
        }

        public int DeleteUser(int Id)
        {
            using (var connection = GetDbConnection())
            {
                return connection.Execute(@"Update [user] SET IsActive = 0
WHERE UserId = @Id", new {Id        });
            }

       }
    }
}
```

Program 5. *Implementation of general methods in the user repository*

5.3 Controller with dependency injection

After completing the repository's implementation, they are ready to be used from the other layers to execute CRUD operations. The repositories classes can be accessed from the controller class to communicate with the data access layer. In order to access

the repository's methods, we need to instantiate a class with a "NEW" keyword as Program 6.

```
public class controller
{
   public void Update (int id)
   {
       UserRepository userRepository = new UserRepository();
       userRepository.GetUserById(id);
   }
}
```

Program 6. *Building dependency by accessing the user repository from the controller*

In this way, we are creating a dependency between the controller class and the repository class. By creating dependencies in the components, we lose flexibility. In other words, in the case of reimplementing or having multiple implementations of repositories, it is not possible to swap out the dependency easily and it makes the classes be dependent on each other. [38]

To avoid dependency between classes and respect the loose coupling we need to implement the Dependency Injection into our controller. Dependency injection (DI) is a design pattern that allows loosely coupling of the software components and makes it possible to manage the maintainability and complexity of source code.

The dependency injection happens in three different ways: Constructor injection, property injection and setter injection. Here the easiest and the most popular type of DI, namely, constructor injection is implemented. In the constructor DI, a dependency is passed through the constructor of the class. [39] The controller class with the injected repositories is written as Program7:

```
public class UsersController
        {
            private readonly IUserRepository userRepository;
            private readonly IRoleRepository roleRepository;

            public UsersController(IUserRepository userRepository, IRoleRe-
pository roleRepository)
            {
                this.userRepository = userRepository;
                this.roleRepository = roleRepository;
            }
        }
```

Program 7. *Accessing repositories using dependency injection*

In the next step, we need to pass in the concrete implementation of the IUserRepository from where the controller is called. When the number of dependencies increases, using

a framework is beneficial to handle them and decreases the complexity of the manual work. Here we have chosen the Ninject framework which is lightweight and easy to operate. [40]

After setting up the Ninject framework, it configures the dependencies at run time as Program 8:

```
 private static void RegisterServices(IKernel kernel)
    {
        kernel.Bind<IUserRepository>().To<UserRepository>().InSingle-
tonScope();
        kernel.Bind<IRoleRepository>().To<RoleRepository>().InSingle-
tonScope();
    }
```

Program 8. *Configuring dependencies using Ninject framework*

The next move is to add the action methods into the controller. The controller is responsible for answering the requests coming from the view, so the action methods in the controller are responsible for returning the results [41]. Therefore, initially, we need action to create a list of all the users from different databases. The users can be fetched from the new table (IAM) that has been illustrated in the first section. As all the users are now in a single place, we just need to inject the corresponding aggregate to the controller. Afterward, the user's repository provides the list of users to the method.

The IAM table contains both user's ids from different user's tables of different databases. Therefore, it acts as a link where the user's information from the original tables can be fetched. The user's list view model is the iamDto class. Therefore, the action returns the iamDto object to the view as Program 9.

```
public ActionResult UsersList()
        {
            var userInfo = userProvider.GetUserInfo();
            var userListItemViewModels =
                iamUserRepository
                .GetAllIamUsersForMRPOrCMCompanyId(userInfo.ActiveMRPCompa-
nyId,         userInfo.CMActiveCompanyId)
                .Select(item => new UsersListItemViewModel
                {
                    IamUserDto = item
                }).ToList();

            return View(userListItemViewModels);
        }
```

Program 9. *Implementation of get request in controller*

Once the list of users is exposed in the main view of the user management tool, we need to implement the user modification functionalities. The edit user view model contains the user information which is stored in different tables. For example, user roles and rights which are fetched from a different table than the newly created IAM table. As explained earlier, the new table keeps the user IDs of original tables as MrpUserId and CmUserId. In this way, the user's relations to the other tables, in particular roles table are still valid.

Since the edit view model has information about different tables, it is necessary to map the view model to each repository DTO object. The edit view model uses the IAM table user-id as user identification. Therefore, we need to fetch the users from each database by their corresponding user IDs. This will happen by calling read functions in each repository. Afterward, we can apply the updated information to each repository in different databases.
There are several places where the user's data can be modified as stated before. Hence, it is preferred to centralize the create, delete and edit operations in a commonplace. In this way, these actions can be reached from the other projects and any changes to the mechanism of these functions would only apply in a singular place. In order to accomplish this task, we are using the Service layer.

5.4 Service layer

The repository layer gives abstraction over the data access layer and exposes the CRUD operations. All the database logic goes to the repository for separation of concerns. For example, listing the users as UsersList function is implemented in the repository and the user repository has all the information needed for accessing data. The controller uses the repository layer in the action methods and does not contain any database logic. Hence, separation of concerns appears, as repositories are responsible for data access logic, and the controller is in charge of controlling the logic of application flow. In this case, we need to take a position for business logic.

Aforementioned, the CRUD operations should happen against the tables in different databases at the same time to persist the consistency. On the other hand, there are a few tools and features which are manipulating the user's data. Considering this situation, a commonplace is a good solution to centralize user data modification. One option is using a service layer. The Service layer refers to the part of the system which is located in the middle of a multi-tier architecture and mediates the interaction between the controller and the repository layer. The service layer can be used for validation and other business logic operations [42].

The identity access management service layer interface is implemented as Program 10:

```
namespace Fatman.Service
{
    public class IamUserAggregate
    {
        public IamUserDto IamUserDto { get; set; }
        public MRPUserDto MRPUserDto { get; set; }
        public CMUserDto CMUserDto { get; set; }
    }

    public interface IIamUserService
    {
        IamUserAggregate CreateUser(IamUserDto iamUser);

        IamUserAggregate UpdateUser(IamUserDto iamUser);

    }
}
```

Program 10. *Implementation of service layer interface*

The implementation of the update user method is written Program 11:

```
public IamUserAggregate UpdateUser(IamUserDto iamUserDto)
        {
            var result = new IamUserAggregate();

            result.IamUserDto = _iamUserRepository.Update(iamUserDto);

            if (iamUserDto.CmUserId.HasValue)
            {
                var ecenterUserDto =
                    _CmUserRepository.GetCMUserBy-
CMUserId(iamUserDto.CmUserId.Value);
                if (CmUserDto != null)
                {
                    CmUserDto.Email = iamUserDto.Email;
                    CmUserDto.Username = iamUserDto.Login;
                    CmUserDto.Name = iamUserDto.FirstName + " " +
iamUserDto.LastName;
                    CmUserDto.Phone = iamUserDto.Phone;
                    /..
                }
                result.CmUserDto = _CmUserRepository.UpdateC-
mUser(CmUserDto);
            }
            if (iamUserDto.MrpUserId.HasValue)
                {
                    var MrpUser = MrpUser.MrpUserReposi-
tory.GetUser(iamUserDto.MrpUserId.Value}).FirstOrDefault();

                            MrpUser.Email = iamUserDto.Email;

                            MrpUser.FirstName = iamUserDto.FirstName;

                            MrpUser.LastName = iamUserDto.LastName;
```

```
                            MrpUser.Login = iamUserDto.Login;
                            //..
                    result.IamUserDto.MrpUserId = MrpUserReposi-
tory.SetUser(MrpUser);
                    }

            return result;
        }
```

Program 11. *Implementation of the updateUser method in the service layer*

The service should be injected into the controller in order to talk to the service layer. In this way, the business logic is moved to the service layer and the service layer isolated from the controller. As a result, it is possible to reach the service layer's methods from any application and apply the changes to the user's data in a unique way.

Program 11 demonstrates one example feature of editing user basic information from the database layer to the controller. The other functionalities required for the administrator of the user management tool is developed with the same logic. The repository classes are implemented for relevant aggregates of each application such as users, roles, rights, filters, and companies.

5.5 Unit testing

The repository pattern makes it easier to unit test the application. The unit testing refers to a software testing method, in which any individual component of the software is tested. These units are the smallest testable piece of software. The aim of unit testing is to ensure that each component of the system performs as designed [43].

After implementing the required features, in order to test the repository's performance, we will write some unit tests. By implementing the repositories in the domain layer, the application layer is not directly dependent on the infrastructure layer where the repositories are implemented. As a result, after making the dependency injections to the controller, it is possible to create the mock repositories.

As can be seen from Program 12, the Mock object is a fake object which simulates the behavior of a real object. The Mock repository returns the fake data rather than the real data from the database. This is useful when the real data are impractical to be manipulated. [44]

For the purpose of writing the unit testing, a new test project with dependency on the repository's implementation is created. Thereafter, the test class and the mock repository object is implemented as Program 12:

```
 public class UserRepositoryTest
    {
        private readonly IUserRepository repository;

        public IamUserRepositoryTest()
        {
            var repositoryMock = new Mock<IUserRepository>();
            repository = repositoryMock.Object;
        }
}
```

Program 12. *Implementation of a mock repository*

Following that, it is possible to inherit the mocked repository and use it in the classes for each method being tested. For example, for the Create method in User repository we have the class Create as Program 13:

```
public class Create : UserRepositoryTest
        {
            private readonly string login;

            public UserDto user { get; }

            private readonly UserDto resultUser;

            public Create()
            {
                login = $"UnitTest{DateTime.UtcNow}";
                user = new UserDto(login:login, frameCompanyId:2);
                resultUser = repository.Create(user);
            }
}
```

Program 13. *Create a function in the user repository*

As can be seen, the mocked user is created, and the resulting user is ready to be tested. In testing, the readability should be considered as one of the most important aspects. The reason is that the expected result should be explicitly extracted from the code reading. Hence, the method names should represent the scenario of what being tested and the expected result out of it. This also aids in providing documentation. Thereby, only by reading the name of the test, the behavior of it is interpreted. In this manner, whenever the test is failed, it is easy to understand what scenario does not fulfill the expectations.

As an illustration, some test methods on user creation are provided as Program 14:

```
        [Fact]
        public void UserShouldBeReturned()
        {
            Assert.NotNull(resultUser);
        }

        [Fact]
        public void UserShouldHaveAnId()
        {
            Assert.True(resultUser.Id > 0);
        }

        [Fact]
        public void UserActiveIsTrue()
        {
            Assert.True(resultUser.LastLoginIsSet);
        }
```

Program 14. *Implementation of unit test methods on user creation*

6. EVALUATION

This chapter evaluates the implementation of user authentication and user management tool. The outcome is evaluated by reflecting on how well the business requirements and system requirements are met. In addition, the compatibility of the results with the company's future plans is discussed.

6.1 Meeting thesis goals

In terms of meeting the thesis objectives, a new mechanism for user authentication and authorization is implemented by utilizing the microservices paradigm. The implementation of user authentication and authorization is followed by developing a new user management tool.

As stated in the previous chapters, the new federated identity management is implemented in a separate application without attempting to make the existing source code more complex and harder to maintain. Componentizing the authorization and authentication responsibility is inspired by handling the microservices cross-cutting concerns. Accordingly, similar methods for the implementation of an identity provider is conducted.

In the new system, it is managed to implement a new mechanism of authorization and authentication, in a manner that new tools and applications can be added to the current products. The new tools and applications can be added to the comprehensive system without worrying about the existing services and user login, similar to microservices architecture.

This approach can be developed further without interfering with monolithic applications utilizing modern technologies. Migration to the microservices architecture model is a trend for modernizing the existing monolithic applications [45]. Accordingly, the new mechanism of user authentication facilitates the architectural decisions in the future model of Fatman products to migrate to the microservices model.

The user management tool is implemented by following the domain-driven design approach. The separation of concerns in the design of the tool is accomplished in the implementation Which also refers to a crucial principle in microservices.

The new user management tool allows the unit testing against the database which is not feasible in other data access patterns. This feature is beneficial in particular for user

modification functionalities due to the importance of user's and the company's sensitive information.

6.2 Meeting business requirements

The business requirements reflect the decisions made by management on enhancing the company's products. The marketing plan of Fatman business has come to a conclusion of offering unique software to the market. The idea of having maintenance resource planning with the addition of a contract management system is a unique offer on the table of enterprise companies.

Having the federated identity management aids the company to make this offer feasible.

However, the implementation of federated identity management is not adequate to fulfill the business requirements. As explained before, the lack of user access control causes the overload of work to the service desk and development which is costly for the company. The new user management tool eliminated the number of task tickets to these teams caused by the mentioned issues.

6.3 Meeting functional requirements

In respect to the functional requirements, the users of both applications do not need to attempt for logging several times in the new unified system. The access to both applications is given by a single login which eliminates password fatigue, potential user errors and time consumption.

In the new application, the whole system is presented as a uniform while there are separated subsystems under the hood. From the end-user perspective, the style and design of subsystems are matched and moving between the subsystems is possible through shared sidebar navigation. As a result, moving between the subsystems is user-friendly and easy to operate.

Moreover, the administrators of software are able to create or modify users in the comprehensive application. In the new tool, it is possible to give access and rights of each intended subsystem to the user. Hence, the user management of the new unified system is feasible without Fatman service help desk interference.

Given the unified user management tool, the user data is consistent and changing the information from any feature of the program applies in all the corresponding tables simultaneously.

The entire additional features are implemented in a reasonable time frame that fulfills the customer's requirements, enhances the user experience and accordingly improves client satisfaction.

6.4 Meeting nonfunctional requirements

The implementation of the identity provider to handle the authorization and authentication of the integrated system is made in a separate component. Therefore, this component can be scaled and maintained independently.

Encapsulating this feature in a separate module prevents making the existing code more complicated and larger. The separate component can be modified without affecting the other applications. It is also possible to deploy this new module separately.

Having the identity provider enhances the performance and user experience of the system. By managing to make the separate applications work together, a new future model for building and adding new tools is attained. In this way, it is possible to add the new features and tools to the existing system as a separate component with the preferred technology selection.

7. CONCLUSION

The central goal of this thesis was to design a mechanism that enables the federation of user authentication, authorization, and management of different monolithic applications. These monolithic applications are legacy business software that has been growing over the years. Accordingly, developing additional critical requirements for these systems makes the maintenance and scalability of the system more challenging. Consequently, we were seeking a solution that allows the development of user authorization and management of our business systems utilizing new technologies.

As stated in chapter three, in this research it is attempted to concentrate on the development of the tasks in preference to gathering the theoretical solutions. Therefore, in this thesis firstly the development of the new features is accomplished and secondly, the evaluation against the wellness of the features is performed.

The challenges of implementing the new user management tool and user authorization and authentication are known as cross-cutting concerns. Based on the existing researches and own analysis, the cross-cutting concerns of Fatman products are similar to the concerns of adapting to the microservices architecture model. In this regard, it has been decided to utilize the microservices model to enhance the development of new mechanisms. This approach can be followed further in the future plans of the company to modernize the whole architectural design of the system by moving to the microservices.

In order to follow the microservices principles, firstly, the similarities of both systems are clarified and secondly, the applicability of the comparable solutions is discovered. The main characteristics of microservices that aids us in this project are associated with the componentization by services and decentralized data management. In this regard, the main concepts and solutions to data management and user authentication are extracted from microservices and utilized in designing the corresponding tools.

After establishing methods to achieve the accomplishment of a federated identity management system, the implementation of a new user management tool is demonstrated in detail. The implementation of the tool initiated by illustrating the Fatman product's high-level database architecture and primary user data mapping between the databases. The implementation followed by demonstrating a lower layer of data accession. Afterward, the implementation of business logic is explained according to the design. Eventually,

the implementation of unit testing against the database is demonstrated to assure each unit is functioning as designed.

In respect to following the design science approach, it is required to make the evaluation against the performance of implemented artifacts. In this context, the list of various requirements and objectives of the thesis is recorded, and the wellness of each result is clarified.

2. REFERENCES

[1] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, “Microservices,” *IEEE Softw.*, vol. 35, no. 3, pp. 96–100, 2018.

[2] “Fatman.” [Online]. Available: https://fatman.fi/en/. [Accessed: 07-Nov-2019].

[3] M.Rouse, “What is legacy application? - Definition from WhatIs.com.” [Online]. Available: https://searchitoperations.techtarget.com/definition/legacy-application. [Accessed: 02-Nov-2019].

[4] M.Fowler, “Microservices.” [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 02-Nov-2019].

[5] D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation,” *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 22–32, 2017.

[6] “What is an ESB (Enterprise Service Bus)? | IBM.” [Online]. Available: https://www.ibm.com/cloud/learn/esb. [Accessed: 07-Nov-2019].

[7] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, “Microservices in Practice, Part 1: Reality Check and Service Design,” *IEEE Softw.*, vol. 34, no. 1, pp. 91–98, 2017.

[8] M.Fowler, “PolyglotPersistence.” [Online]. Available: https://martinfowler.com/bliki/PolyglotPersistence.html. [Accessed: 02-Nov-2019].

[9] “Multi-tenant SaaS patterns - Azure SQL Database | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/azure/sql-database/saas-tenancy-app-design-patterns. [Accessed: 02-Nov-2019].

[10] M.Fowler, “Decentralized Data Management.” [Online]. Available: https://martinfowler.com/articles/microservices.html#DecentralizedDataManagement. [Accessed: 02-Nov-2019].

[11] A. El-Hokayem, Y. Falcone, and M. Jaber, “Modularizing behavioral and architectural crosscutting concerns in formal component-based systems – Application to the Behavior Interaction Priority framework,” *J. Log. Algebr. Methods Program.*, vol. 99, pp. 143–177, 2018.

[12] M.Ayoub, “Microservices Authentication and Authorization Solutions.” [Online].

Available: https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a. [Accessed: 02-Nov-2019].

[13] A. Furda, “On Multitenancy, Statefulness, and Data Consistency,” *IEEE Softw.*, pp. 63–72, 2018.

[14] A. C. Michalos and H. A. Simon, *The Sciences of the Artificial*, vol. 11, no. 1. 1970.

[15] S. TAKEBAYASHI, T. HASHIGUCHI, S. TANAKA, and S. ISHIZU, “on Botryoid Sarcoma (Mesodermal Mixed Tumor) in the Perineal Region in Infant.,” *Gan No Rinsho.*, vol. 10, pp. 501–509, 1964.

[16] P. Jausovec, “What are sticky sessions and how to configure them with Istio? - DEV Community ????” [Online]. Available: https://dev.to/peterj/what-are-sticky-sessions-and-how-to-configure-them-with-istio-1e1a. [Accessed: 13-Nov-2019].

[17] “High Availability with Sticky Session Replication | Jelastic Dev Docs.” [Online]. Available: https://docs.jelastic.com/session-replication. [Accessed: 11-Nov-2019].

[18] S.Peyrott, “What Is and How Does Single Sign-On Authentication Work?” [Online]. Available: https://auth0.com/blog/what-is-and-how-does-single-sign-on-work/. [Accessed: 02-Nov-2019].

[19] P.Otemuyiwa, “JSON Web Tokens vs. Session Cookies: In Practice.” [Online]. Available: https://ponyfoo.com/articles/json-web-tokens-vs-session-cookies. [Accessed: 02-Nov-2019].

[20] “JSON Web Token Introduction - jwt.io.” [Online]. Available: https://jwt.io/introduction/. [Accessed: 02-Nov-2019].

[21] C. Guo and Y. Wang, “Application of federated identity management in ERP system,” *Proc. 2008 IEEE Int. Conf. Serv. Oper. Logist. Informatics, IEEE/SOLI 2008*, vol. 2, pp. 1971–1974, 2008.

[22] “OpenID Connect | OpenID.” [Online]. Available: https://openid.net/connect/. [Accessed: 02-Nov-2019].

[23] K. K. and M. K. A. Banati, E. kail, “Authorization and Authentication orchestrator for micoservice-based software architetures,” pp. 1180–1184, 2018.

[24] “OASIS Security Services (SAML) TC | OASIS.” [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security. [Accessed: 02-Nov-2019].

[25] “OAuth 2.0.” [Online]. Available: https://oauth.net/2/. [Accessed: 02-Nov-2019].

[26] K.Casey, “Understanding OAuth 2.0 and OpenID Connect — Runscope Blog.” [Online]. Available: https://blog.runscope.com/posts/understanding-oauth-2-and-openid-connect. [Accessed: 02-Nov-2019].

[27] “Dev Overview of OpenID Connect.” [Online]. Available: https://developers.onelogin.com/openid-connect. [Accessed: 02-Nov-2019].

[28] P.Otemuyiwa, “SSO Login: Key Benefits and Implementation - DZone Security.” [Online]. Available: https://dzone.com/articles/sso-login-key-benefits-and-implementation. [Accessed: 02-Nov-2019].

[29] O. Zimmermann, “Microservices tenets: Agile approach to service development and deployment,” *Comput. Sci. - Res. Dev.*, vol. 32, no. 3–4, pp. 301–310, 2017.

[30] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead,” *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, 2018.

[31] E. Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software: Amazon.de: Eric J. Evans: Fremdsprachige Bücher,” vol. 7873, no. 415, p. 529, 2003.

[32] M.Fowler, “BoundedContext.” [Online]. Available: https://martinfowler.com/bliki/BoundedContext.html. [Accessed: 02-Nov-2019].

[33] “Repository Pattern A data persistence abstraction.” [Online]. Available: https://deviq.com/repository-pattern/. [Accessed: 02-Nov-2019].

[34] M. Fowler, *Patterns of Enterprise Application Architecture*, 1st editio. Addison-Wesley Professional, 2002.

[35] “Designing the infrastructure persistence layer | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design. [Accessed: 02-Nov-2019].

[36] M.Fowler, “P of EAA: Data Transfer Object.” [Online]. Available: https://martinfowler.com/eaaCatalog/dataTransferObject.html. [Accessed: 02-Nov-2019].

[37] “Dapper ORM Tutorial and Documentation.” [Online]. Available: https://dapper-tutorial.net/. [Accessed: 02-Nov-2019].

[38] “Dependency Injection.” [Online]. Available: https://www.tutorialsteacher.com/ioc/dependency-injection. [Accessed: 02-Nov-

2019].

[39] I.Ilin, “All Dependency Injection Types.” [Online]. Available: https://medium.com/@ilyailin7777/all-dependency-injection-types-spring-336da7baf51b. [Accessed: 02-Nov-2019].

[40] “Ninject - Open source dependency injector for .NET.” [Online]. Available: http://www.ninject.org/. [Accessed: 02-Nov-2019].

[41] “Dependency injection into controllers in ASP.NET Core | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-2.2. [Accessed: 02-Nov-2019].

[42] “Validating with a Service Layer (C#) | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/models-data/validating-with-a-service-layer-cs. [Accessed: 02-Nov-2019].

[43] P. Hamill, *Unit Test Frameworks: Tools for High-Quality Software Development*. 2009.

[44] “Best practices for writing unit tests - .NET Core | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices. [Accessed: 02-Nov-2019].

[45] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture,” *IEEE Softw.*, vol. 33, no. 3, pp. 42–52, 2016.

3. APPENDIX A: SCRIPT FOR MAPPING USERS OF DIFFERENT DATABASES USING SQL

```
with MRP_CTE
    ([Login],
    [column1],
    [column2],
    [column3])
    as (
    Select
    [Login],
    [column1],
    [column2],
    [column3],
    MRPConnectionString
    from (
    SELECT
      fui.[Login]
      ,c.[column1]
      ,fu.[column2]
      ,fu.[column3]
      ,lower ('Server=' + c.[Server] + ';Database=' + c.[Dbase]) as MRPCon-
nectionString
  FROM
    [MRP_AuthorizationDB].[dbo].[UserInfo] as fui
    join [MRP_AuthorizationDB].[dbo].[User] as fu on fui.ID = fu.UserInfoID
    join [MRP_AuthorizationDB].[dbo].[UserCompany] as uc on fu.UserID =
uc.UserID
    join [MRP_AuthorizationDB].[dbo].[Company] as c on uc.CompanyID= c.ID
  )
   SQ
)
,
CM_CTE ([Login], [column1], [column2], [column3])
AS
(
   SELECT [username], [column1], [column2] , [column3], CMConnectionString
from (
    SELECT
     eu.[username]
      ,eu.[colum1]
      ,eu.[column2]
      ,eg.[column3]
      ,lower ('Server=' + ead.[server] + ';Database=' + ead.[dbase]) as
CMConnectionString

  FROM [CM_AuthorizationDB].[dbo].[user] as eu
            join [ECENTER].[dbo].[group] as eg on eu.groupid = eg.id
            join [ECENTER].[dbo].[company] as ec on ec.id = eg.companyid
            join [ECENTER].[dbo].[accessDescription] as ead on ec.[de-
faultCID]=ead.cid
    ) as SQ
)
```

```
,
CM_only_CTE (CM_Login, CM_column1, CM_column2, CM_column3, CM_Connection-
String, MRP_Login, MRP_column1, MRP_column2, MRP_column3, MRP_Connection-
String)
as (
    select CM.[Login] CM_Login
           CM.[ConnectionString] CM_ConnectionString,
           MRP.[ConnectionString] MRP_connectionString,
           MRP.[Nickname],
           MRP.[GroupID],
           MRP.[WorkTitleId]
           from CM_CTE e
        left join MRP_CTE f on CM.[Login]=MRP.[Login] and MRP.[Connection-
String]= CM.[ConnectionString]
        where MRP.[Login] is null
)
,
MRP_and_both_CTE (CM_Login, CM_FirstName, CM_LastName, CM_Email, CM_Connec-
tionString, MRP_Login, MRP_FirstName, MRP_LastName, MRP_Email, MRP_Connec-
tionString)
as (
    select CM.[Login] CM_Login
           ,CM.[FirstName] CM_column1
           ,CM.[LastName] CM_column2
           ,CM.[ConnectionString] CM_ConnectionString
               ,MRP,[Login] MRP_Login
           ,MRP.[FirstName] MRP_column1
           ,MRP.[LastName] MRP_column2
           ,MRP.[ConnectionString] MRP_connectionString

    from MRP_CTE f
        left join CM_CTE e on CM.[Login]=MRP.[Login] and MRP.[Connection-
String]=CM.[ConnectionString]
)

,
iam_user_cte (CM_Login, CM_column1, CM_column2, CM_ConnectionString,
MRP_Login, MRP_column2, MRP_column3, MRP_connectionString)
    as (
        select * from CM_only_CTE
            union all
        select * from MRP_and_both_CTE
        )

 INSERT INTO [MRP_AuthorizationDB].[iam].[User]
           (     [Login]
        ,[column1]
        ,[column2]
        ,[column3]
           )

     select
        case when MRP_Login is not null then MRP_Login else CM_Login end
        ,case when MRP_column1 is not null then MRP_column1 else CM_column1
end
        ,case when MRP_column2 is not null then MRP_column2 else CM_column2
end
        ,case when MRP_column3 is not null then MRP_column3 else CM_column3
```

```
end
  from iam_user_cte
```