Expertise
and insight
for the future

Irina Voloshina

Applying microservices pattern to monolithic software

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Information Technology

Bachelor's Thesis

26 October 2020

Metropolia
University of Applied Sciences

Author Title Number of Pages Date	Irina Voloshina Applying microservices pattern to monolithic software 33 pages 26 October 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of School Kari Juvonen, Product Owner Tero Suominen, Managing Director

Cloud computing and microservices patterns have changed the ways in which business processes are modeled and implemented. New possibilities have emerged to modernize legacy applications.

This project studied the practical application of microservices and breaking down the process of transition from monolithic structure to microservices using a legacy project from the case company. An evaluation of a complex project was performed and a roadmap for restructuring and modernization using microservices was created. The evaluation identified existing flaws and restrictions, defined goals for the transition and explored the ways to benefit from shifting to the microservices architectural pattern.

The motivation for this project for the case company was to try out the microservices and study their application by decomposing the legacy software to solve the problems related to the monolithic structure of the application. Due to a high degree of coupling between the code components adding new features or making significant changes to the project in accordance with business logic changes was difficult. A roadmap to transition is offered as an alternative.

As a result of the study, potential challenges and benefits of applying the microservices pattern to the case project were evaluated. The application was broken down into models according to the principles of the domain-driven design to aid the transition. The roadmap for the transition with the specific technological stack was presented to the case company as well as the alternative solutions to overcome the challenges. The study concluded that the case project is likely to benefit from the transition to microservices despite the possible challenges and minor drawbacks.

Keywords	microservices, monolithic, software architecture, domain-driven design, architectural pattern

Contents

List of Abbreviations

1 Introduction 1

2 Methods and materials 3

3 Literature review 4

3.1 Architectural patterns 4
3.2 Microservices and SOA 6
3.3 Microservices and domain-driven design 7
3.4 Choice of architectural pattern 8

4 Transition to microservices 10

4.1 Potential benefits of microservices 10
4.2 Potential drawbacks of microservices 12
4.3 Database structure breakup 12

5 Case project 16

5.1 Software description 16
5.2 Technological stack and current state 17
5.3 Motivation and expectations 19

6 Redesigning case product 20

6.1 Identifying system architecture 20
6.2 Decomposition strategy 23
6.3 Transition roadmap 24
6.4 Microservices architecture 25
6.5 Potential benefits of transition 27
6.6 Potential challenges/drawbacks of transition 28

7 Conclusion 30

References 32

List of Abbreviations

SOA	Service-oriented architecture
DDD	Domain-driven design
REST	Representational state transfer
API	Application programming interface
IDE	Integrated desktop environment
ESB	Enterprise service bus
CQRS	Command query responsibility segregation
CRUD	Create, read, update, delete (basic database operations)
DBMS	Database management system

1 Introduction

The term “microservices” has been in use since at least 2011 [1]. It is applied to a type of a software architecture which leverages use of separate services to promote scalability, availability and flexibility. A traditional monolithic application is developed as a single item. While the initial deployment is usually simple and straightforward, as the codebase grows and new features are implemented, the development and maintenance of the code can become a daunting task.

In the business context, the more time the developers have to spend on untangling the software challenges, the costlier the software maintenance becomes. The system can essentially become a Big Ball of Mud. [2.]

The emergence of microservices is tightly related to the development of cloud computing. Cloud computing and introduction of an economy of scale have solved some of the challenges businesses have traditionally encountered, such as availability, scalability and making digitalization significantly more cost-effective. [3, p.3.]

Microservices is a type of architecture which aims to solve challenges of monolithic applications. Microservices are complete pieces of functionality that are constructed along the business capabilities, providing clearer view on the functionality compared to monoliths. Since one of the most important characteristics of microservices is independent deployability, the development of microservices can be delegated to multiple teams at the same time, increasing flexibility and speed of development.

Although microservices have become increasingly popular, this architectural pattern is not applicable or beneficial to each and every project. While providing certain benefits, there are trade-offs for using microservices. Therefore, before deciding on the architecture for a new project or upgrading legacy software, the benefits and potential drawbacks have to be carefully weighed.

This thesis provides a proof-of-concept regarding transition from the monolithic application to the microservices architecture. The software in question is Reha, a complex resource management system that has been in use for over 20 years. The case company

selected the project for transformation since the update in the software technology stack is highly desired. The software consists of multiple tightly coupled domains and functionalities. By untangling the project structure into microservices, the most desired benefits are increased availability, reduced maintenance costs and added flexibility for the development.

The outcome of the study is a plan on rebuilding the application's architecture by using microservices. This plan also includes an evaluation of potential benefits and drawbacks, as well as specific alternatives to the currently used technological stack. It divides the structure of the project into separate business domains and provides suggestions on the process of the transition.

The thesis consists of the following chapters:

- Chapter 2, "Methods and materials", briefly explaining the phases and the workflow of the project.

- Chapter 3, "Literature review", presenting theoretical findings on architectural patterns (monolith and microservices), comparing them and explaining relationships between microservices and other popular concepts – SOA and DDD.

- Chapter 4, "Transition to microservices", analyzing potential benefits and drawbacks of using microservices, as well as tackling the issue of breaking up the database of a monolithic application.

- Chapter 5, "Case project", providing a detailed description on the case application, its usage, current state and technological stack.

- Chapter 6, "Redesigning case product", presenting an evaluation of the application's potential transition to microservices and providing suggestions on tools and technologies as well as evaluating potential benefits and drawbacks for the project.

2 Methods and materials

The thesis project was carried out in several phases. The case company was presented with questions regarding the challenges related to the application use and development. The target software was researched from the perspective of different user groups in order to create a preliminary map of the application functionality without in-depth code analysis. The purpose of this functionality mapping was to create a high-level domain structure of the application in order to use it later for detailed mapping of functionality which would roughly correlate with potential microservices.

An in-depth analysis of microservices was conducted to outline the potential benefits and drawbacks of using microservices. The findings were used as a foundation for the evaluation of the impact that transition to microservices would have on the case project. The evaluation presents the most likely benefits and drawbacks, as well as suggestions to counteract these drawbacks and/or minimize the negative effects from them.

Based on the application map and the evaluation, a roadmap for the application transition to microservices is presented. This roadmap includes specific examples of technologies and methods for transforming the application, with considerations for the development practices in the case company.

3 Literature review

3.1 Architectural patterns

This section describes the monolithic and microservices architectural patterns, their differences and challenges.

A monolithic application is mainly characterized by a high degree of coupling between the code components or having no distinguishable components. The monolithic architecture can be further divided into single-process monolith, modular monolith and distributed monolith architecture. Most of the monoliths are single-process monoliths. In these monoliths, the code is deployed as a single process. The modular monolith is a subtype of the single-process monolith since it allows independent development of separate modules. However, it lacks the independence of microservices; the deployment has to be carried out as a single process. The modular monolith can be considered as one step closer to microservices since the code is divided into modules based on functionality or business logic. [4.]

A content management system, such as Drupal, is an example of a modular monolith: while having relatively independent modules, the system needs to be deployed altogether when the code changes are made. The modules can have dependencies on each other; therefore, not all modules are strictly independent components. [5.]

Nevertheless, modular monolithic architecture is a sufficient choice for a large number of projects. It can serve as an intermediate step in the transition from monolith to microservices. It is also important to consider factors such as the size of application and the potential for application growth. If the application is meant to be small, microservices might only unnecessarily complicate its structure without providing any significant improvements.

Monolith architecture can bring a number of benefits to the application. The development of monolith architecture is more straightforward since it is not distributed; it also allows relatively simple developer workflows, including end-to-end testing (which is more complicated in distributed systems), troubleshooting and monitoring.

While being suitable for smaller projects, monolith architecture often presents challenges once the codebase starts to grow. It is difficult to establish code ownership and therefore define clear responsibilities for the teams; the codebase becomes tangled and complex, with tight coupling and low cohesion [4], which slows down development process; the changes made to an application require deployment of the whole system; scaling can become inefficient. If in microservices often only particular service(s) need to be scaled, in a monolith the whole application needs to scale [6]. If different parts of a monolithic application are, for example, resource-intensive, the requirements for server configuration are higher and might be costlier. In addition to this, introducing new technologies to monolith might be close to impossible while in the microservices pattern each service may use a different stack and the only thing they need in common is a means of communication.

Microservices can be defined as follows: “independently deployable services modeled around a business domain” [4]. This definition describes the main idea of microservices – having independently deployable components. The more expanded definition states the following:

> Microservices is an application architectural style with a clear bounded context, interface and dependencies emerged from the Domain-Driven Design architectural patterns and DevOps; each microservice is a loosely coupled service with a single responsibility principle, where each component is a full but miniature application that’s focused on producing a single business task [7].

Business domain is a concept widely used in Domain-Driven Design to describe a business process or a set of business processes which serve a single purpose and can be developed by a single team for a specific group of users [7]. A service can be defined as follows:

> A service is a standalone, independently deployable software component that implements some useful functionality [6].

An alternative to implementing a service is implementing a library. Generally, it is recommended to implement a library only for the functionality that is unlikely to change since a library can be used in multiple services and change in a library might have an effect on all the services using the given library.

Figure 1 represents the difference in scaling between a monolith and microservices.

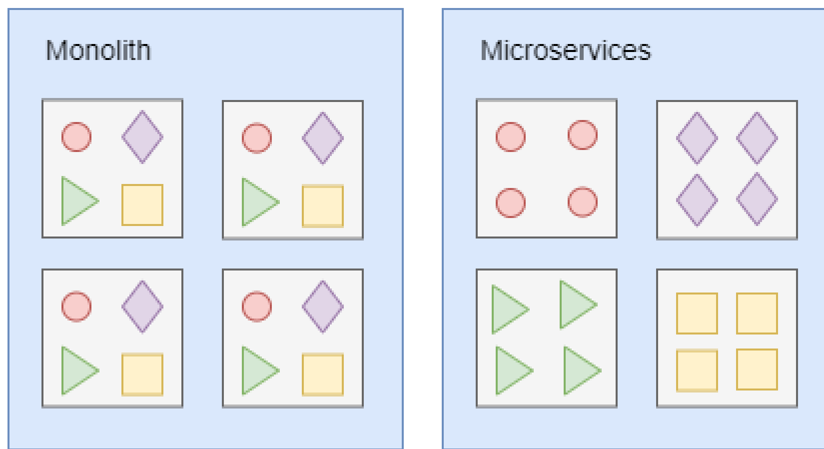

Figure 1. Scaling: monolith vs. microservices. Adapted from [4].

3.2 Microservices and SOA

Service-oriented architecture (SOA) was introduced a long time before microservices, in the late 1990s. While sharing multiple similar traits, these two terms are not interchangeable. Like the goal of microservices, the goal of SOA is to create a set of loosely coupled components while hiding complexity of the underlying architecture. However, the SOA primarily serves a specific business function while microservices are constructed on a different basis.

Newman describes microservices as a type of SOA [4]. However, some authors consider microservices to be different from SOA. There is a number of differences between SOA and microservices, such as scope and communication, that these authors consider sufficient to create a clear distinction between SOA and microservices. While SOA applies enterprise-wide, comprising the whole system, microservices are applied within the scope of an application. However, drawing of the service boundaries is subjective [4]. Another difference is that SOA is usually tightly coupled with the business function it caters to and includes all the code and required external integrations. This is because the primary function of SOA is to deliver a service for a business function. SOA leverages component reuse; in microservices, extensive reuse of the service might end up in excessive dependencies and tight coupling, reducing agility. [8.]

Methods of communication are also different for SOA and microservices: while enterprise resources are generally available through synchronous calls, microservices leverage the

use of asynchronous calls since synchronous communication leads to real-time dependencies and coupling and therefore decrease in resiliency. Another significant difference is usage of an ESB (enterprise service bus) in SOA, which is a common communication mechanism for all services. This type of architecture significantly reduces availability since ESB becomes a single point of failure. Microservices are typically being kept as small as possible while SOA applications can be complex and large. [8.]

3.3 Microservices and domain-driven design

Microservices commonly utilize the concepts and logic of Domain-Driven Design. Domain-Driven-Design is a software development approach revolving around connecting the complex business model to the implementation. It is a framework with strategic value the objective of which is to map business domain concepts to software artifacts. [7.]

Following the context of DDD, the strategic planning of transitioning an application to microservices has the following steps:

1. **Create high-level domain model of the application**. This model consists of the largest domains and subdomains within the application which are mapped to the real-world aspects of a solution [7]. This model gives an evaluation of the areas the application spans. The domain model helps to capture the specifics of each area, including the concepts, requirements and processes related to the domain.

2. **Define bounded contexts**. Bounded context is a pattern in DDD that groups models within the domain by the common features, such as ubiquitous language and intercommunications between these models [9]. The models within bounded contexts are unified and the vocabulary used within the bounded context is the same for all models. Often bounded contexts are the scope of a single team; however, in the case project there is only one small team of developers, so the bounded contexts do not carry as much importance in the case project.

3. **Define entities, aggregates and services**. Entities and aggregates can be considered building blocks in the tactical approach to DDD. Entity is defined as a mutable (subject to state change) noun with its own intrinsic identity that cannot

be shared but that can be associated with other entities. An important feature of entities is history, meaning that the entity and operations performed upon it are traceable.

4. **Identify microservices**. Microservice is a logical unit of functionality which takes in specific data, processes it and emits certain data to the next service. Microservices have logic boundaries and defined interfaces which allow them to communicate to other services. After the bounded contexts, models and aggregates have been identified, a set of microservices can be outlined with each one having a narrow functionality related to entities or aggregates. Microservices typically belong to one bounded context.

3.4 Choice of architectural pattern

As mentioned earlier, the microservices are not the ultimate solution for all the projects and systems. As an example, small projects do not require breaking up into smaller pieces; the benefits of microservices do not apply in small projects since they make the structure more complex. A small application benefits from the monolithic architecture due to the following characteristics:

- simple to develop;
- radical changes are easy to implement;
- straightforward testing process;
- straightforward deployment process;
- easy to scale.

However, for bigger applications microservices are also not necessarily the pattern of choice. The architecture of an application should be planned in accordance with the business processes and logic of the system. If microservices fail to serve the purpose of an application or do not reflect the business processes, their use should be reevaluated. In

other words, microservices by themselves are neither a goal nor an automatic win; the goal should lie in the area of business processes and business goals; microservices are only a tool that possesses a potential to aid in achieving the goals. When deciding on whether to use microservices, the following aspects should be considered:

- What is hoped to be achieved;
- Have other alternatives been considered;
- Metrics and milestones to measure the effect of transition.

The need for decomposition can be based on facilitating the division of labor and knowledge to benefit from specialized knowledge and skills to be productively applied to the respective parts of an application.

Microservices correspond to business capabilities. A large application with a variety of domains and business models, with distributed teams and/or great need for availability and scalability is likely to benefit from microservices.

4 Transition to microservices

4.1 Potential benefits of microservices

Potential benefits from using microservices include the following:

- **Improved team autonomy**. While shifting more responsibility onto the team can be achieved in other ways than changing an architecture, adopting microservices architecture creates a space for a team's autonomy. The potential benefit of improving team autonomy is empowering people and giving them a place to grow and develop both individually and as a team.

- **Reduced time to market**. Since microservices are small independent components of architecture owned by relatively small teams, the changes in the application are performed only where and when necessary. Teams can work autonomously on their assigned services without the need to redeploy the whole application. Nevertheless, there are numerous other factors that can be involved in the software delivery process, meaning that low speed of shipping can, in fact, have reasons going beyond the architectural style of an application. Therefore, while microservices can help reduce time to market, all steps in the software delivery process should be inspected. [4.]

- **Cost-effective scaling for load**. Microservices allow to break an application into multiple parallel processes which can in long-term perspective improve the overall performance of an application as well as allow selective scaling, i.e. scale only the processes which are under load in a given period of time. However, this specific goal can be achieved without shifting the architecture by either vertical or horizontal scaling and using a load balancer, which is native for cloud-based applications. [4, 6.]

- **Improve robustness**. Microservices help to decouple and decompose the processes into independent operations within the application in a way that a failure in one process does not lead to halt of the whole system. As with the scaling, a certain degree of robustness can be achieved with horizontal scaling behind the

load balancer and spanning multiple copies of an application over different failure plans (such as having the machines in different physical locations). By adding redundancy, robustness is improved without the need to change architecture.

- **Scale team size**. As microservices are independent parallel processes, adding more developers can accelerate the delivery speed since the teams work independently and do not require interaction with other teams. They do not need to wait for any other team to do the work first; all work can be done in parallel. The more tasks are done in parallel, the more developers can be thrown to the project to execute the tasks independently. However, for this pattern to work, the collaboration between teams and the codebase/service ownership have to be planned and implemented carefully.

- **Adopt new technology**. Due to independence of microservices, they are not required to share the same technology stack. The inner structure of microservices are hidden from the outside world; the interfaces on the borders of microservice need to be unified. This is one of the major differences between monolith and microservices; monolith requires the same technology or set of technologies to be used across the whole application since the system is deployed as a whole. From another point of view, applying new technology to services of limited size makes it easier to learn and understand the technology in isolation, i.e. it shortens the path of adopting new technology not only for a given application, but for the future projects as well. Ease of adoption allows to experiment more frequently and cost-efficiently compared to trying to implement an unknown technology into a big wholesome project.

- **Code reuse**. While reusing code is considered to be a good practice, it is not a goal. Rather, code reuse is a convenient feature of microservices which leads to other benefits while not being a benefit as-is. Code reuse can help to satisfy other goals, such as quicker shipping of features, and reduce time-to-market by speeding up the development. However, code reuse should be used cautiously to avoid creating unnecessary dependencies between services and increasing code coupling.

4.2 Potential drawbacks of microservices

While individual service is easy to develop, test and deploy, the complexity of the whole system grows with the number of services. One group of issues that arise from this complexity is related to monitoring, log segregation, troubleshooting and exception tracking [6]. Since each service normally produces its own logs and traces, the issue of untangling the log information emerges. Therefore, some sort of logging solution needs to be implemented.

End-to-end testing becomes more complex since it spans multiple services. It is considered good practice to use automated testing for individual services. Even though end-to-end testing is not recommended, testing still must ensure that the services work together as expected. [4.]

Since microservices operate over the network, the communication is not instantaneous; it means, that some degree of latency is introduced into the application. The problem of latency is especially significant when an application performs, for example, a complex database transaction that spans multiple services or processes messages from the event queue, where the order of transaction steps or messages are important [4, 6]. Therefore, it is an additional task for the developers to implement application logic in a manner that would prevent data loss and/or ensure the order of events and steps is maintained. Due to the nature of communication between the services, some packets of data can be lost; if the service is unavailable, synchronous calls to this service will fail. This is why an asynchronous model of communication is generally preferable between microservices.

4.3 Database structure breakup

Unlike most monolithic applications, microservices use multiple databases as each service owns only a specific set of data. Therefore, simple ACID transactions cannot be implemented when the service needs to update other service's data. This poses the challenge of keeping the data consistency across services. There are two ways to solve this problem: distributed transactions and sagas. Sagas are the preferred method of dealing with database updates for a number of reasons. First of all, the distributed trans-

actions are not supported by modern message brokers, such as Apache Kafka or RabbitMQ. Another issue with distributed transactions is reduced availability since their functionality requires synchronous communication. The more services are involved in the distributed transaction, the less the availability is, since the availability of each service is less than 100%. [4, 6.]

While Newman [4] only advises not to use distributed transactions (2PC), Richardson [6] strongly discourages their use in favor of sagas. In addition to this, Richardson recommends the usage of the CQRS (command query responsibility segregation) pattern.

The CQRS pattern is an approach to separate read and update operations for any given data storage. Traditionally, read and update operations are handled by the same data model, which might include complex validation on update operations and complicated queries on retrieving the data for read operations. Often, read and write workloads in the database are asymmetrical and therefore require a different amount of resources and scaling. [10.]

A saga is a form of asynchronous pattern for performing transactions in multiple services, maintaining the data consistency. The transactions performed in each service are simple ACID transactions. These local transactions are coordinated in a saga using asynchronous messaging. From the definition of asynchronous communication, sagas do not require all participating services to be online at the same time. Sagas are the preferred pattern for microservices usage. Nevertheless, there are challenges associated with sagas. One of the challenges is transaction rollback. The rollback of an ACID transaction is an automatic database operation; rolling back a saga requires separate rollback transactions. In other words, a saga requires two sets of transactions: for committing the change and for reverting the changes (compensating transactions). Compensating transactions are applied in a reverse order. If for an **n** number of transactions, T_n fails, the transactions T_1 to T_{n-1} must be rolled back. The compensating transactions start at C_n (compensating for T_n) and go down to C_1 (compensating for T_1). Not all steps necessarily require rollbacks; read operations do not cause any change in the database and therefore no compensating step is needed. [6.]

The transaction steps in a saga are divided into three groups:

- **Compensatable transactions** – the transactions followed by the steps that can fail, therefore most likely requiring compensating mechanisms for a case of failure;

- **Pivot transaction** – a transaction followed by a step that never fails (by design);

- **Retriable transactions** – the transactions that always succeed (by design).

Saga's transactions can be coordinated in two ways: using choreography or orchestration. Choreography is a decentralized method of data coordination with its participants directly exchanging information with each other. Orchestration is a centralized method of saga coordination. A saga coordinator sends command messages to services to perform local transactions. These ways are similar to brokerless (orchestration) and broker-using (orchestration) methods of asynchronous communication.

Choreography-based sagas use publish/subscribe model of communications. The events are published to corresponding channels. The services listen to the events in order to initiate the transaction. The services do not need to know about each other's location. Therefore, choreography sagas support loose coupling. However, sagas can contain cyclic dependencies and have to be implemented carefully. Also, depending on the workflow within the saga, there is a risk of tight coupling between services. According to Richardson [6], simple sagas can be implemented with choreography; more complex sagas benefit from orchestration.

Orchestration-based sagas use the command/async reply model of communication. An orchestrator sends a command to a service and then processes the reply in order to determine the next step. Using orchestration-based sagas promotes less coupling since the services do not need to be aware of each other's existence or even of the existence of the saga orchestrator since all the coordination logic belongs to an orchestrator. In this context an orchestrator can be considered as a separate service. However, Richardson [6] recommends using orchestrators with caution and avoiding putting business logic into an orchestrator as it can increase coupling and centralization.

A major drawback of using sagas is a lack of isolation. All the transactions that happen in saga are instantly committed and visible to other services and sagas. Therefore, other

services and sagas can alter the data involved. Nevertheless, a saga possesses atomicity, consistency and durability features. In return for lack of isolation, sagas usually provide higher performance.

Lack of isolation in sagas can occasionally cause abnormal database behavior, known as anomaly. The anomalies can be classified as following:

- Lost updates: a saga overwrites some data which was previously altered by another saga;

- Dirty reads: a transaction or a saga reads the updates made by a saga that has not yet completed those updates;

- Fuzzy/non-repeatable reads: two different steps of a saga read the same data and obtain different results because another saga has made updates.

When using sagas, lack of isolation should be addressed with countermeasures. It is a responsibility of the developers to design sagas in a way that would prevent anomalies or at least minimize their impact on business processes. [6.]

5 Case project

5.1 Software description

The Reha application is a software application for enterprise resource management. The application is designed to be used by administrators and employees which are considered as resources. The resources in the application are **employees** and **machines**.

The resources can be assigned to specific places or tasks, such as:

- workstations;
- job locations;
- orders;
- workgroups;

The main business processes in the application are related to resource management. Examples of these business processes include:

- allocating employees to orders/tasks based on highest available competence required for the task;
- allocating machines to orders/tasks;
- managing shift schedules and allocating employees on work shifts;
- reporting task completion;
- resource usage statistics and analytics;
- salary calculation based on the work performed, employee competence class and specific rules;

- machine lifecycle management, including scheduled maintenance and repairs;
- keeping track of employee vacations, sick leaves, overwork allowance and other absences.

5.2 Technological stack and current state

The underlying technology of the Reha product is CA Plex. CA Plex is a tool for Architected Rapid Application Development. The key features of CA Plex are:

- Model-driven development. Systems are designed with business requirements in mind.
- Patterns.
- Dynamic inheritance and customization. Changes to patterns are applied throughout an entire model.
- Code generators. RPG III/IV, Java, C++, C# Server, with options for .NET Client and EJB support.
- Industry standard database support. Extensive DB support, including SQL Server, Oracle and DB2 for i.
- Web client development and Service-Oriented Architectures (SOA). Patterns available for AJAX/RIA systems and SOAP/XML components.
- Configuration management with versioning facilities.
- Application integration. Integrates with CA Technologies and third-party tools.

CA Plex provides an IDE with the graphical user interface for the development. Developers work with blocks and panels in the designer; the code of the application is automatically generated. The generated code is native for the platform it was done for. [11]

There are two integrations to external services:

- Crystal Reports for statistics and analytics;
- Silta payroll system.

The system is hosted on IBM i servers.

IBM i is an operating environment developed by the IBM company which includes database, file serving, security, virtualization, work management, networking and connectivity, auditing, storage management and a number of other capabilities. This operating environment is most commonly used for secure and resilient resource-intensive business applications. [12.]

There are several issues related to the IBM i platform itself. While being recognized for its scalability, security, reliability and outstanding upwards compatibility, IBM i is often perceived as outdated. The fact that IBM i originated in 1988 only supports this perception. Nevertheless, IBM i servers are widely used for business-critical applications and as a crucial part of the whole computer infrastructure in many organizations. However, the IBM i talent is gradually becoming less and less available, posing a serious problem for those organizations. [13.]

Lack of available skills pushes organizations to switch to other systems and technologies. In the case reason, this reasoning is partially behind the motivation to give up the usage of IBM i servers for the Reha infrastructure.

At the starting point, the product has been fully developed, with occasional bug fixes. The responsible team consists of two professionals sharing equal responsibility for the project as a whole. The installation of the product for the customer is handled by the product owner in the case company. The update process is performed using IBM I savf and ZIP C++. All the customers using a given product have used it for many years and during at least last year there were no customer support requests. Both customers and developers do not pinpoint any challenges related to the use and development of the product. The product is a standalone system with an integration to an external third-party salary management software.

There are different versions of the Reha application. The first version consists of a C++ client installed on customer machines and the IBM i server featuring the AS/400 system and an integrated DB2 database. This version was developed in the 1990s and is currently in active use.

The second version consists of a C++ client on customer machines, the IBM i server and a Microsoft SQL Server database. This version has been in production testing for a customer, but has not replaced the first version completely.

The third version of Reha features a web client. Essentially, it is an original Reha version that was adapted to the cloud with no significant changes to its functionality.

5.3 Motivation and expectations

There are several reasons behind transforming the case product into microservices. First of all, the case company plans to give up the use of IBM i servers for a given application. Secondly, the modernization of the technological stack is required. The case company also considers the application to be an educational example of implementing microservices since this pattern is unfamiliar.

The updated application should be secure and resilient; the installation process should be simplified; onboarding of new developers should be easy and quick. It is important that the application is available and scalable since processes at times are resource-intensive.

6 Redesigning case product

6.1 Identifying system architecture

Richardson [6] suggests that the first step in defining an application structure is to define system operations. Defining system operations means taking into account user stories, system use scenarios and application requirements. The result of this identification is a high-level domain model. One of the techniques to perform this is to derive the domain model from the nouns of user stories (such as Order, Salary, and Workstation) and the system operations from associated verb phrases (such as place order, calculate salary, and request resource for the workstation). The system operations affect the domain objects, their content and the relationships between these objects.

On the highest level, the system operations in Reha can be globally divided into the following:

- user management;
- employee management and allocation;
- machine management and allocation;
- workstation management;
- job locations management;
- order management and allocation;
- salary management (including integration with third-party service);
- gathering and processing statistics using Crystal Reports;
- administrative tasks (global application settings);

- extensive information queries.

The high-level domain model does not represent the final version of the implementation; nevertheless, it helps to create vocabulary to describe the system operations in the application. The Reha project domain model and associated concepts are presented in figure 2.

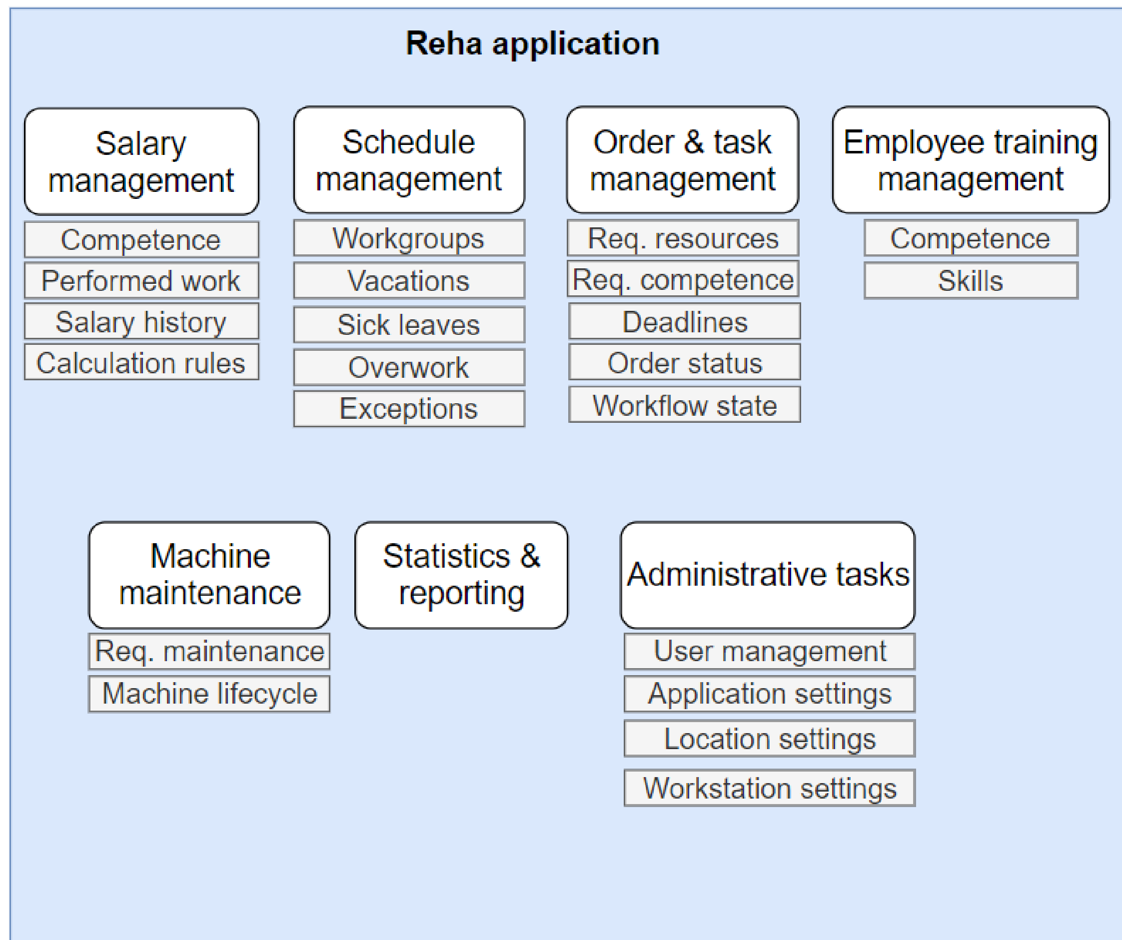

Figure 2. Domain model and model-related concepts of Reha application

From there the user stories associated with the domain objects can be expanded to greater detail. For example, expanding the employee management associated scenarios reveals additional entities to be used in the application:

- An **employee** always belongs to a **workgroup**. A workgroup defines the schedule for the employee.

- An **employee** is also associated with the **training**. Training increases employee competency in different areas and therefore influences the salary.

The above user stories suggest existence of at least two additional entities: workgroup and training. Other entities are presented in figure 3.

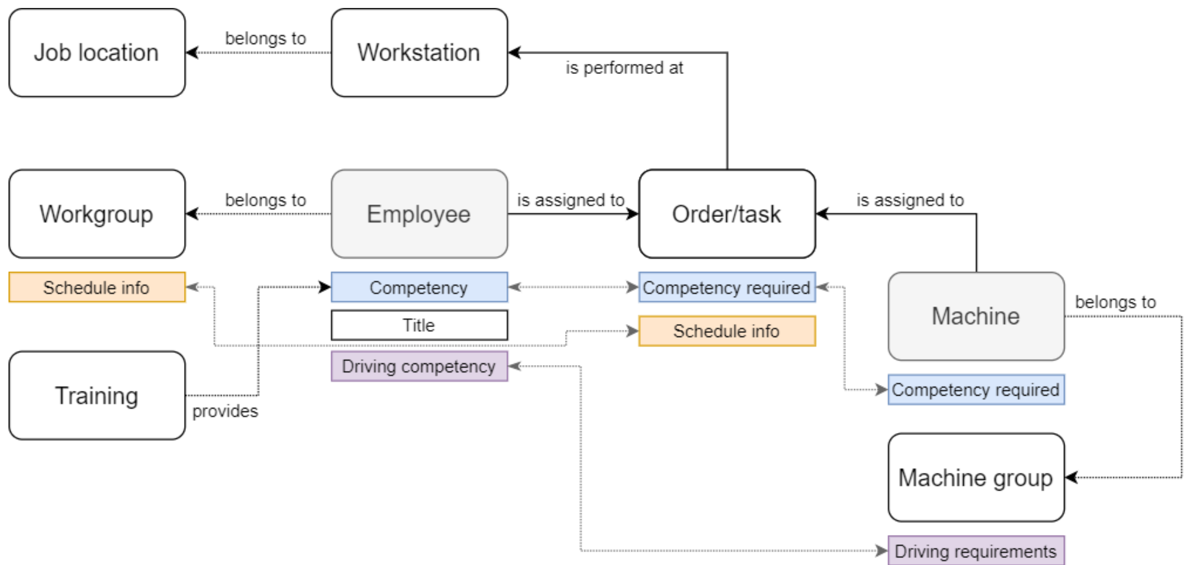

Figure 3. Domain entities in Reha application

The next step is to identify the requests within the application.

The system operations can be divided into commands and queries. Commands are operations that create, update and delete data. Queries are operations that read data. Both command and query are terms related to CQRS [6, 10]. The system commands are the ones that are primarily identified by the verbs from the user stories. Each system operation's behavior specification includes the preconditions, which must fulfill for the operation to invoke, and post-conditions, which must be true after the operation was invoked. Richardson [6] points out that commands are more architecturally relevant, though the queries role is not completely insignificant.

For example, the list of commands for an employee entity looks like the following:

- Employee's competency is updated after receiving the training.

- Employee can be created in the system or deleted; employee's personal and professional information can be updated.

- Employee's vacation and other leaves can be created, updated and deleted.

- Employee can mark task as completed, and the completed task contributes to employee's salary.

- Employee can be assigned to a training, workstation, job location or workgroup.

- Employee can have non-machine equipment assigned (e.g. gloves, robe, tools)

These commands, their pre- and post-conditions need to be taken into account when planning the microservice architecture. In the same manner the requests are identified for other domain entities (listed in figure 3).

6.2 Decomposition strategy

There are several approaches to the decomposition. One approach is to outline services on the basis of business capabilities. Another approach is to use domain models outlined in an earlier step as a foundation for defining services. [6.]

In this project, the strategy selected for decomposition is based on the domain model. This strategy can be applied to this project since there are clearly defined entities (figure 3) upon which certain actions are performed. In fact, the development in CA Plex happens in a way that is to some extent similar to creating domain models. Terms "model", "object", "entity" and "library" are widely used in CA Plex; therefore, the developers can refer to the current application structure for reference when laying out the updated structure of the application based on domain models [14].

To decompose a database, it is vital to determine which data should belong to which service. Often same data can span multiple services. For example, if there is a single service that manages employee data and another service that manages the orders, both services need to know specific information about an employee. The information must be

up-to-date and accurate; otherwise, an employee might end up being assigned to two different tasks at once. These situations must be prevented by application logic.

Since there are integrations with external software, it would be beneficial to create separate services for interacting with these third-party components. Therefore, there is a separate service for the Silta integration and a separate service for feeding data to Crystal Reports.

There is also some functionality that does not necessarily require a service; rather, it can be used as a library since the data it handles rarely changes and the data it contains is simple. Thus, attributes such as employee competence classes, list of tools and non-machine equipment can be transformed into separate libraries that can be used by a variety of other services.

On the basis of the domain entities, the data ownership has to be established for all the data in the Reha database.

6.3 Transition roadmap

The great role in applying the microservice pattern to a legacy application is strategic planning of both the desired outcome and the process of transition itself.

There are various approaches to the transition. Some approaches are based on isolating specific related functionalities one by one and separating them from the rest of the codebase in order to be later replaced with an updated service. However, in the case project the most likely approach is to recreate the application with an updated technological stack that would allow for easier installation, update and scaling processes.

The microservices can generally be tricky to implement on the software that is completely shipped to a customer since the installation requires technical skills. Nevertheless, one solution for such software is to use containerization. Using container services such as Docker provides a possibility to develop microservices independently to later be installed on customer machines. Another benefit of containerization is having a platform-agnostic application, meaning that the application is able to run on any platform provided [15].

The technological stack recommended for the given project:

- container system, e.g. Docker;
- ASP.NET Core framework with Entity Framework Core;
- Microsoft SQL Server or MySQL databases (might vary based on the service);
- cache system, e.g. Redis cache;
- Swagger for API specifications [16];
- hosting either on Azure infrastructure or on-premises on the customer machines;

The recommendation is based on the specifics of the software usage and the developers' skills. The team responsible for the product development has extensive knowledge of Microsoft services and technologies. The cache system is recommended since the amount of data in the system is large and there is a significant number of queries that are run in the system which potentially span multiple services. Having unified well-defined API simplifies communication between services and version control of different APIs, ensuring their compatibility.

6.4 Microservices architecture

The suggested architecture is only based on the initial evaluation; nevertheless, it can be used for initial planning for transition. The structure of each individual microservice can also vary.

Generally, figure 4 shows simplified structure of a microservice that can be used for developing simple CRUD services with a selected technological stack.

The service consists of the codebase and a data storage. Depending on the purpose of the service, the data storage can be relational DBMS, NoSQL database or any other king of data storage.

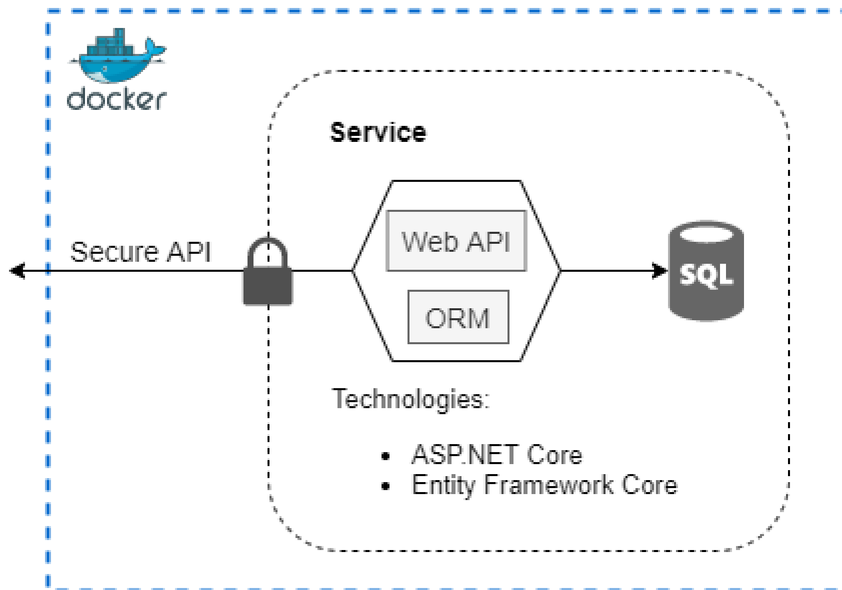

Figure 4. General structure of a microservice in Reha. Adapted from [16].

Entity Framework Core is a lightweight data access technology. It is an ORM that allows developers to work with the database using .NET objects. Data access is performed using an entity class (such as employee) and a derived context, representing a session with a database. [16.]

As an example, employee data in a model within such service could look like this in C#:

```
public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public bool IsFullTime { get; set; }
    public bool IsActive { get; set; }
    public string PhoneNumber { get; set; }
    public LicenceClass DrivingLicence { get; set; }
    public int IsInReha { get; set; }

    // Additional code ...
}
```

Listing 1. Example of employee data in entity class.

The derived context, based on this model, could look like this in C#:

```
public class EmployeeContext : DbContext
{
    public EmployeeContext(DbContextOptions<EmployeeContext> options) :
base(options)
```

```
    { }
    public DbSet<LicenceClass> LicenceClasses { get; set; }

    // Additional code ...
}
```

Listing 2. Example of derived context based on employee data model.

The service can be deployed either together with other services on the same Docker host or separately on a different host.

The communication between the microservices is event-based. Figure 5 represents a possible communication model for the statistics service.

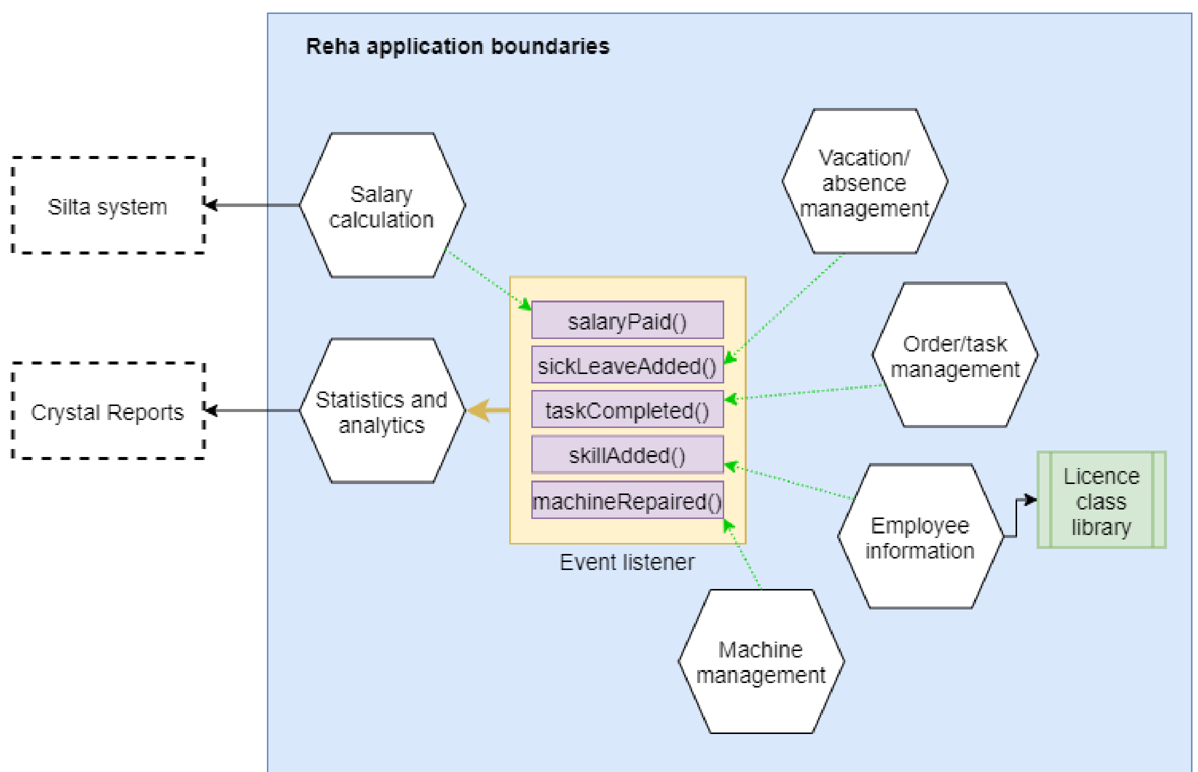

Figure 5. Event-based communication for statistics service

6.5 Potential benefits of transition

Considering the size of an application, development practices and the use cases of the application, the potential benefits from transition to microservices are:

- modernization of the application due to update of the whole technological stack;
- resiliency and availability;
- efficient computing and memory resource usage since application services are scaled according to workload;
- simplified installation process;
- platform-agnostic application, due to containerization;
- faster introduction of new features into the ready product;
- clear view on business domains, functionalities and codebase structure;
- faster onboarding of new developers whenever needed;
- increased speed of database transactions due to usage of cache and sagas;
- simplified side-by-side development of several versions;
- opportunity to introduce new technologies into the application or its parts if needed.

6.6 Potential challenges/drawbacks of transition

While bringing a number of benefits to Reha project, transition to microservices can introduce challenges during/after transition. These challenges are:

- Process of transition is most likely to be lengthy. Given the current state of the application and technological stack in use, as well as the existence of three different versions with various technologies, the transition basically requires rewriting the whole application anew.

- The application development does not benefit from shared responsibility since the development team consists only of two people. In turn, it might initially create significant development overhead. However, the transition creates an opportunity to introduce new developers, creating a more resilient team with a variety of skills.

- The database breakup process and redesigning the database transactions with the use of sagas requires extensive planning. Critical processes need to be identified and implemented in a way that would minimize a possible negative impact on business logic.

- The application processes need to be redesigned with consideration to introduced latency between services and implement measures that would minimize the effect of latency or packets loss on business logic.

When planned and implemented carefully, microservices are more likely to be a beneficial pattern for the Reha project. However, it requires extensive planning and restructuring.

7 Conclusion

Transformation of a legacy application with the use of microservices is a way to make an application more modern, resilient and cost-effective. In addition to this, the microservices can be beneficial for the development process and the maintenance of an application, improving teams' autonomy and significantly reducing the time to market. However, the microservices are not a universal solution for every project.

Microservices and monolithic architectures are both valid choices when selecting an application structure. There are both benefits and drawbacks to each pattern. The choice of application structure requires thorough analysis of the contributing factors, such as application complexity, size and use cases. If the transition is considered, an evaluation of possible benefits and drawbacks should be carried out to obtain a general idea whether an application would benefit from the change.

The microservices pattern is closely related to the service-oriented architecture although these concepts are not interchangeable. Generally, the microservices pattern can be considered a service-oriented architecture with the smaller scope and the greater focus on service independence.

The transition to microservices is a rather complex procedure which requires extensive planning and preparation. The planning phase of the transition can be carried out using the concepts of the domain-driven design. Breaking down the application structure into domains, models and entities helps to determine the data ownership and the boundaries for each service in the application, as well as their interconnections. The structure breakup is performed both in the code and in the database.

The Reha application was developed in the 1990s using CA Plex. The motivation of the case company for applying the microservices pattern to this application was primarily to study the impact of microservices on a legacy application and update in Reha's technological stack. The study determined the transition roadmap for the application and suggested a technological stack with the use of containerization and Microsoft-based technological stack. The development team's skill stack is largely based on Microsoft technologies, and there are numerous technologies in the Microsoft technological stack for the development of microservices applications specifically.

According to the analysis and evaluation, the Reha project is likely to benefit from microservices architecture since microservices will help to untangle the complicated structure of the project, providing clear view on the project functionality and separating unrelated functionalities, increasing the general availability and scalability of the product. Using containerization can simplify the installation process since the product will not need to be modified to comply with any possible underlying system. The transition roadmap takes into account the skills and capabilities of the development team as well as the current development processes to make the transition less costly while largely beneficial for the application.

The next step for the Reha application transformation is to define the domains and the models within the project structure for each component. Since CA Plex uses the similar terminology to the one of the domain-driven design, this task is not too complex. Once the models are defined, the service boundaries can be drawn and the service communications can be outlined. The technological stack for an individual service depends on the purpose of the service, its relation to other services and the requirements such as scalability, availability, latency and resiliency.

Upon the completion of the transition, the case company can observe the performance of the application, the performance of the development team and analyze the customer experience in order to measure the impact of the microservices on the Reha application.

References

1 Fowler, Martin; Lewis, James (25 March, 2014) *Microservices.* Available at: https://www.martinfowler.com/articles/microservices.html (Accessed 10 October, 2020)

2 Foote, Brian; Yoder, Joseph (1999) *Big Ball of Mud*. Available at: http://www.laputan.org/mud/ (Accessed 12 October, 2020)

3 Sajee, Mathew (2020) *Overview of Amazon Web Services*. Available at: https://docs.aws.amazon.com/whitepapers/latest/aws-overview/aws-overview.pdf (Accessed 14 October, 2020)

4 Newman, Sam (2019) *Monolith to Microservices*. O'Reilly Media, Inc. Available at: https://learning.oreilly.com/library/view/monolith-to-microservices/9781492047834 (Accessed 12 September, 2020)

5 Young, Margy Levine et al. (2020) *Overview of Drupal*. Available at: https://www.drupal.org/docs/understanding-drupal/overview-of-drupal (Accessed 14 October,2020)

6 Richardson, Chris (2018) *Microservices Patterns.* Manning Publications. Available at: https://learning.oreilly.com/library/view/microservices-patterns/9781617294549 (Accessed 10 October, 2020)

7 Norelus, Ernese (28 April,2019) *Implementing Domain-Driven Design for Microservice Architecture*. Available at: https://medium.com/design-and-tech-co/implementing-domain-driven-design-for-microservice-architecture-26eb0333d72e (Accessed 18.09.2020)

8 IBM Cloud Team (02 September, 2020) *SOA vs. Microservices: What's the Difference?* Available at: https://www.ibm.com/cloud/blog/soa-vs-microservices

9 Fowler, Martin (2014) *Bounded Context*. Available at: https://martinfowler.com/bliki/BoundedContext.html (Accessed 16 October, 2020)

10 Microsoft Docs (11 February, 2020) *Command and Query Responsibility Segregation (CQRS) pattern*. Available at: https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs (Accessed 18 October, 2020)

11 Broadcom (2014) *CA Plex Data Sheet.* Available at: https://docs.broadcom.com/doc/ca-plex (Accessed 17 October, 2020)

12 Perry, Trevor (21 December, 2019) *What Is This Thing Called IBM i?* Available at: https://freschesolutions.com/whats-thing-called-ibmi/ (Accessed 17 October, 2020)

13 Huntington, Tom (16 January, 2020) Is the AS/400 Dead? HelpSystems Blog. Available at: https://www.helpsystems.com/blog/as400-dead (Accessed 26 October, 2020)

14 CA Plex Documentation. Available at: https://techdocs.broadcom.com/us/en/ca-enterprise-software/intelligent-automation/ca-plex/7-2-1.html (Accessed 26 October, 2020)

15 *Docker overview*. Available at: https://docs.docker.com/get-started/overview/ (Accessed 17 October, 2020)

16 Microsoft Docs (14 August, 2020) *Creating a simple data-driven CRUD microservice*. Available at: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/data-driven-crud-microservice (Accessed 17 October, 2020)