ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

###### Факультет компьютерных наук

Мамонтова Мария Евгеньевна

ФИО

Конструктор корпоративных приложений на основе микросервисов

**Название ВКР**

**Группа \_\_\_\_\_\_\_\_\_**

Выпускная квалификационная работа - МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

по направлению 09.04.04 «Программная инженерия»

Программа: Системная и программная инженерия

|  |  |
| --- | --- |
|  | Научный руководитель  Зыков С.В.  Доцент департамента программной инженерии факультета компьютерных наук |
| Москва 2017 | |

FEDERAL STATE AUTONOMOUS EDUCATIONAL INSITUTION FOR HIGHER PROFESSIONAL EDUCATION NATIONAL RESEARCH UNIVERSITY

«HIGHER SCHOOL OF ECONOMICS»

###### Faculty of Computer Science

Maria Mamontova

Name, Surname

Microservice-based enterprise application builder

**Title**

**Group \_\_\_\_\_\_\_\_\_**

Qualification paper – Master of Science Dissertation

Field of study 09.04.04 «Software engineering»

Program: System and Software Engineering

|  |  |
| --- | --- |
|  | Supervisor  Sergey V. Zykov  Associate Professor  [School of Software Engineering](http://cs.hse.ru/en/dse/)  [Faculty of Computer Science](http://cs.hse.ru/en/) |
| Moscow 2017 | |

**Abstract**

The current document is a master thesis work, which is dedicated to the analysis of architecture frameworks for Enterprise System. The basics of the analysis is Microservices’ Architecture which is a modern approach for cloud distributed system development.

The analysis of Microservice Architecture allows determining Microservice weak and strong sides, consideration of the frequently used Microservices patterns and construction architecture solution for microservices management platform.

The topicality of the work is caused by the novelty of the architectural style, which is currently ubiquitously discussed and developed. The application of the cloud oriented microservices methodology to the Enterprise systems particularly worth attention. Recently such application was impossible to imagine, however today enterprise cloud based solutions --- due to the improved security techniques and modern approaches, such as private clouds.

Key words: Enterprise System, Enterprise Architecture, Microservices, Microservice Architecture, Pattern.

Graduation work: 37 pages, 33 pictures, 3 tables, 21 sources

**Аннотация**

Данный документ является магистерской диссертацией, которая посвящена анализу архитектурных каркасов корпоративных приложений. Основным объектом анализа была выбрана Микросервисная архитектура, которая является современным подходом к разработке распределенных систем с применением облачных технологий.

Анализ микросервисной архитектуры позволяет выявить сильные и слабые стороны подхода, рассмотреть часто-используемые при разработке микросервисов шаблоны и построить архитектурное решение платформы для создания приложения на основе микросервисов.

Актуальность работы подтверждается новизной архитектурного стиля, на данный момент методология микросервисов активно обсуждается и развивается. Особенно интересным является использование методологии микросервисов с применением облачных технологий для создания корпоративных систем. Совсем недавно данный подход казался невозможным для корпоративных приложений, но теперь, благодаря усовершенствованным методам защиты информации и новым подходам, таким как частные облака, облачные решения для корпративных систем стали востребованными.

Ключевые слова: корпоративные приложения, архитектура предприятия, микросервисы, микросервисная архитектура, шаблоны проектирования.

Дипломная работа: 37 с., 33 рис., 3 табл., 21 источников литературы

Contents

[**1.** **Introduction** 6](#_Toc498897074)

[**1.1.** **Monolithic architecture** 6](#_Toc498897075)

[**1.2.** **Service Oriented Architecture** 7](#_Toc498897076)

[**1.3.** **Microservices** 7](#_Toc498897077)

[**1.4.** **Topicality and Problematic** 9](#_Toc498897078)

[**1.5.** **Business scope** 10](#_Toc498897079)

[**1.6.** **Goals and objectives** 10](#_Toc498897080)

[**2.** **Resources Review and analysis** 11](#_Toc498897081)

[2.1. Microservices characteristics 11](#_Toc498897082)

[2.2. Enterprise Applications in the Clouds 11](#_Toc498897083)

[2.2.1. SAAS, PAAS and IAAS 12](#_Toc498897084)

[2.2.2. Continuous delivery and Continuous Integration 13](#_Toc498897085)

[2.2.3. DevOps 13](#_Toc498897086)

[2.3. Microservices characteristics analysis – тут старая инфа 13](#_Toc498897087)

[**2.4.** **Patterns** 16](#_Toc498897088)

[2.4.1. API Gateway Pattern 16](#_Toc498897089)

[2.4.2. Load Balancer 17](#_Toc498897090)

[2.4.3. Microservices Inter-process communication patterns 18](#_Toc498897091)

[2.4.4. Database patterns 20](#_Toc498897092)

[2.4.5. Service discovery pattern 24](#_Toc498897093)

[2.4.6. Patterns Comparison 24](#_Toc498897094)

[3. Existing EHS enterprise Web Application solution 27](#_Toc498897095)

[**3.1.** **Business Scope** 27](#_Toc498897096)

[**3.2.** **Advantages of the current solution** 29](#_Toc498897097)

[**3.3.** **Disadvantages of the current solution** 29](#_Toc498897098)

[**3.4.** **Current solution performance assessment** 31](#_Toc498897099)

[4. Microservices solution for EHS enterprise Web Application 32](#_Toc498897100)

[**4.1.** **The motivation** 32](#_Toc498897101)

[5. Client – Server Communication 35](#_Toc498897102)

[6. Inter-process communication 36](#_Toc498897103)

[7. Dev-Ops 37](#_Toc498897104)

[8. Results 38](#_Toc498897105)

[9. Future development 39](#_Toc498897106)

[10. Summary 40](#_Toc498897107)

[11. Conclusion 41](#_Toc498897108)

[Acronyms and Terms 43](#_Toc498897109)

[Bibliography 44](#_Toc498897110)

1. **Introduction**
   1. **Monolithic architecture**

Monolithic architecture is one of the most commonly applied approaches for building software applications. It is considered to be reliable and relatively simple in implementation. According to one of the definitions, monolithic architecture implies that functionally distinguishable aspects are not architecturally separate components but are all interwoven [1].More specifically, three-tier architectural pattern may also be determined as monolithic due to the reasoning that the business logic is implemented through the usage of a single server and a single database. One of the obvious disadvantages of the monolithic architecture is caused by the tendency of the applications to grow with the enormous speed. Whilst the application grows, the importance of the scalability property becomes more and more considerable. Besides, the size of the application has a significant impact on the complexity of the development, testing and deployment. In order to make any changes the developer should be at least marginally familiar with the whole system functionality. Additionally, when deploying even the smallest change, the whole system is replicated. Such procedure is exceedingly expensive for the company in regards to the resources. In the picture below the process of the monolithic application replication is illustrated [2].

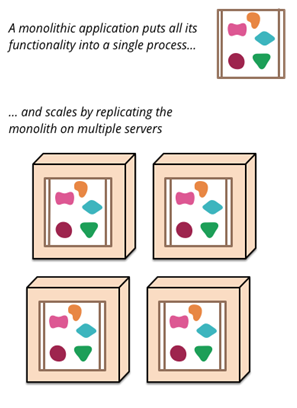


Figure 1.Monolitic application.

Undoubtable advantage of the monolithic application implementation and maintenance simplicity on the earliest stages of the system existence is expelled by the necessity to grow the system on the later stages. In the process of the system maintenance and refactoring the need for a new architectural approach could be required. As one of the possible solutions, monolithic architecture could be transformed into multi-tier application.

* 1. **Service Oriented Architecture**

The methodology that has laid the foundation for the modern distributed systems is Service Oriented Architecture (SOA) (1) .

One of the universal definitions of the term is the following: SOA is a loosely-coupled architecture designed to meet the business needs of the organization [3].

According to the SOA, weakly interconnected services are combined to provide the functionality of the enterprise application, while each service represents some specific function. Consequently, the scalability of the system is supported.

Unfortunately, SOA was not successfully accommodated due to the considerable number of complex abstractions and legacy protocols in methodology. The need to connect many services, written on different languages, has led to the complex Enterprise Service Bus which in turn has led to the archaic and expensive enterprise application, which requires a lot of resources every time the business scope changes or new function is added [4] (Figure 3).

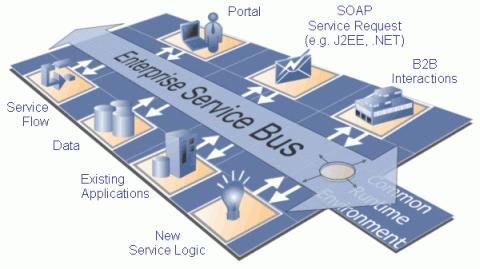


Figure 2. Enterprise Service Bus.

* 1. **Microservices**

SOA has brought a very important idea and partly has become the standard for web applications, but the rapidly changing requirements of the modern systems demanded a new approach. The static models are not viable anymore.

“To deal with the rapidly changing demands of digital business and scale systems up -or down -rapidly, computing has to move away from static to dynamic models. Rules, models and code that can dynamically assemble and configure all of the elements needed from the network through the application are needed.” [5]

The pioneer of the new term “Microservice” Martin Fowler has led the foundation of the numerous discussions, articles and books by his of the same name article called “Microservices -a definition of this new architectural term” [6]. The basic idea of the Microservices correlates with SOA methodology, but Microservices provide clearer vision. Some people call Microservices successful implementation of the SOA. Another opinion states that Microservices describe and implement only a part of the SOA. SOA is a high level Enterprise Architecture methodology, while Microservices are limited by the scope of the system architecture and project management. The correlation between SOA and Microservices is schematically shown in the picture below (Figure 4).

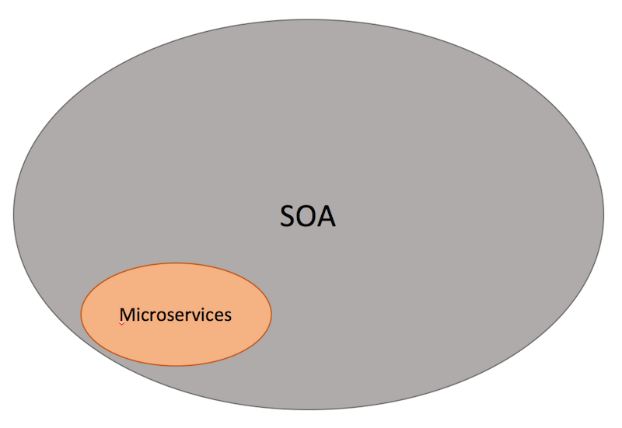


Figure 3. The correlation of SOA and Microservices.

Microservices architecture implies separation of the system into different segments of the business scope, so that each Microservice could work as an independent unit. Microservice methodology reduces heavy Enterprise Service Bus and provides clever endpoints. In order to avoid ponderous integration, which would ruin the independence and scalability, the integration is supported by the usage of HTTP via RESTful APIs, JSON data, and message queues. Microservices are tightly connected with cloud computing, as far as it is absolutely necessary to continuously test and monitor the behavior of the separate microservices and the system overall.

In the picture below, the relation of the system’s parts in software architectures of different times is illustrated (Figure 5).

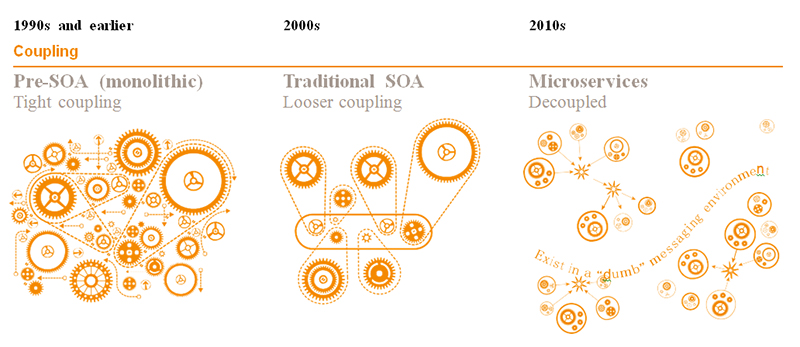


Figure 4. Architecture timeline.

* 1. **Topicality and Problematic**

Martin Fowler’s article was originally published on the 14th of March 2014. Fowler points out that he have not invented a new architectural pattern or style, but generally summarized and formalized the best new practices and described the most distinctive features of the modern trends in the subsequent direction. The pioneers of the style successful implementation are Amazon, Netflix, [The Guardian](http://www.theguardian.com/) and some others.

Discussions and articles has grown into books, the giants of the modern web applications technologies announced the new features for more convenient Microservices implementation and management. For instance, Spring has announced an article about Microservices creation with Spring Boot and Spring clouds [7], Microsoft has provided integration with Docker in Visual Studio 2017 and published many guides on the subject of Microservices implementation, Google has bought the Microservices Api Management platform etc. The term became popular and well-known and the companies started thinking of applying the architecture.

Despite the popularity of the term, the guidelines for the implementation exist mostly in the form of recommendations, which is undoubtedly reasonable. The architecture is not bounded to technologies or languages for implementing the services, there are dozens of patterns for their integration and communication management. The implementation of the notion is unique for each system. Therefore, it is a pressing question for the management and developers team, whether it would be cost-effective to switch to Microservices and what is the best approach for their business scope and initial settings.

* 1. **Business scope**

In the current work the real commercial system will be considered. The system is a monolithic web application for tracking, storing, managing, reporting and analyzing the data of the enterprise in the field of Environment, Health and Safety (EHS ). The initial architecture of the system will be covered in later sections.

* 1. **Goals and objectives**

The goal of the work is to provide the solution for the system migration from the monolithic web application to Microservices, while choosing the most appropriate technologies, patterns and tools.

The objective of the work is to define whether such kind of system would trully benefit from the migration to microservices, how much would it benefit and what the drawbacks are.

1. **Resources Review and analysis** 
   1. Microservices characteristics
   2. Enterprise Applications in the Clouds

The specifics of the Enterprise Applications are the demanding safety requirements. A few years ago, it was impossible to imagine that enterprise activity data could be stored somewhere outside the internal network. However, with the growth of the modern cloud technologies, Enterprise Applications have are mostly cloud oriented. For instance, Microsoft Azure is a common cloud server, used in many enterprise solutions.

There are several options to use clouds for enterprise applications. Enterprise may use Private Clouds, which implies the storage of cloud technologies in the local datacenter. However, similar performance as in public clouds would be very expensive. Public Clouds often provide the Private Clouds Functionality, which considerably reduces the cost. The most effective approach is to use Hybrid Cloud which implies the integration of the local and cloud applications [8].

Let’s compare the traditional on-premises approach with clouds (Table 1).

Table 1.Traditional vs Clouds.

| Characteristics | On-premises | Cloud |
| --- | --- | --- |
| Capacity & Performance | 1. Depends on the hardware 2. Maximal performance load should be beforehand planned and prepared | Unrestricted |
| Cost | 1. High hardware cost 2. The same cost for different performance metrics | 1. Fewer Cost 2. Cost depends on the traffic 3. The more resources need-less the cost. |
| Scalability | 1. Expensive 2. Time-consuming 3. Additional operating efforts | 1. Fewer price 2. Automated scale 3. No operating resources required |
| Continuous Integration & Delivery (Backup, testing, upgrades, maintenance, monitoring) | Manually | Automated |

It may be deduced that Clouds provide faster and cheaper Enterprise Applications operating.

* + 1. SAAS, PAAS and IAAS

In the scope of the cloud services, enterprise may choose the following facilities (Figure 6):

* + - Software-As-a-Service- software, which is working in the clouds. Most frequently, it means, that the vendor provides an access to the application via the web.
    - Platform-As-a-Service- a set of tools which ease development and delivery of the applications;
    - Infrastructure-As-a-Service- provides environment for resources computing, storage and networks management.

Figure 5.SAAS,PAAS and IAAS.

As it was mentioned above, for many enterprises Hybrid Clouds is the most appropriate approach, as it allows to keep some applications on-premises and integrate it with the cloud applications. The logic of refactoring the architecture may be the following – for those applications, which will not evolve, will not have investments or have serious security issues, we choose private On-premises Clouds. For those applications, which are delivered be the vendors, SAAS may be used. Finally, PaaS and IaaS may be used for Cloud Enterprise Applications development and delivery (Figure 7).



Figure 6.Hybrid Clouds.

* + 1. Continuous delivery and Continuous Integration

Nowadays the **implementation** of Continuous Delivery (CD) and Continuous Integration (CI) has become the natural part of the application lifecycle. Both principles underlie Microservice methodology, as far as continuous consistency and availability of the system is very important. Despite the fact that the principles could seem to be similar, there is a difference .

**Continuous Integrations** implies that all development changes should be integrated into the system as soon as possible. The new code snippet integrates into the current versions of the code, usually by making PUSH operations to the Version Source Control. After the code was published, the system should be automatically tested. There are various tools, which detect changes in the code and proceed tests.

Subsequently, Continuous Integration may be logically followed by Continuous Delivery, which provides safe and fast delivery of the software into environment. The deliveries made quite frequently, which helps to support flexible structure of Microservices Architecture. [9]

One famous tool which supports both principles is Jenkins by java [10]. The main functionality of the tool is to proceed predefined steps (build, tests), which are triggered by some event (or time).

* + 1. DevOps

The term DevOps implies the combination of development and maintenance operations and refers to roles and operations which work at the confluence of development and operations. Such Cooperation provides intensive collaboration of the different lifecycle stages and produces high efficiency.

DevOps complies with Microservices methodology. Microservices imply high correlation of all team players through all lifecycle stages. That is also very important for enterprise application, as it allows constantly tracking the needs for changes, immediately response and implementing the required facilities rapidly and coherently [11].

* 1. Microservices characteristics analysis

Despite the fact that Microservices architecture methodology does not have strict rules, schemes or patterns, there are common characteristics. The analysis of the characteristics and their implementation would allow us to understand what could be automated and optimized.

The first characteristic is **Componentization via Services**. Component is a software unit that could be independently replaced and upgraded. The logical decomposition of the system into the components is a task for business analyst and system architect. The representation of the components via services is a task for architect and developer. The mentioned processes are not intended to be automated, but the work on the junction of the different actors’ responsibilities could be proceeded via useful tools.

Therefore, different team players participate on different levels of microservices system construction and implementation. Such participation could be logically and physically interconnected in a single logical space in such a way that the output of one level directly becomes the input of the other level. The interconnection of the processes is shown by the documentation flow provided below (Figure 17).

Figure 7. Documentation flow.

Table 2.Characteristics' analysis.

|  | Description | Managing features | Technical features |
| --- | --- | --- | --- |
| Componentization via Services | The system business scope is divided into individual units, each of which could be implemented by independent Service | -provide the workflow of the inputs/outputs of the lifecycle stages |  |
| Organized around Business Capabilities | Each service should represent single business capability.  The team, which is working on the single Microservice, should be full-stack- beginning from analysts and finishing with managers. | Support the following:  -input business description, documentation of the whole system and of the single service  - assigning Microservice for the specific Business capability.  -Assign people on the microservices  - the stack of the roles for each Microservice -notify user about preferable team size  -statistics about the teams, their work progress, charts and diagrams | -Support integration between the microservices – predefined templates, efficient advices and help on integration choice(questionaries’)  -Provide graphical representation of the system, system interfaces, control inside the team –tasks, code( integration with Version controls systems), interfaces and all other work |
| Products not Projects | A team should be responsible for their product through its full lifetime. This leads the team into everyday contact with their software in production and with their users. | Provide “Support” tool, which is logical continuation of the development cycle , which supports clients appearance on the platform (possible integration with Jira, Zendesk etc) | -Use continuous delivery tools |
| Smart endpoints and dumb pipes | The characteristic implies the reduction of the Enterprise Service Bus and implementation of the work on the end-points of each microservice. |  | -Provide typical patterns implementation (API gateway etc) |
| Decentralized Governance | As far as the system is no longer monolithic and each Microservice represents independent component, there is no need in centralized governance of full value. That means that each team is fully responsible both for their service independent work and integration with other services. | Integrate with continuous integration and delivery tools – this would help to control the system 24/7.  In order to support continuous control it is necessary to provide messaging system, which informs about builds’ statuses. For instance, the system could send messages by e-mail in case of the emergencies and after the planned control. | -Support the different programming languages (integration with git, docker )  -Build new services with consumer driven contracts, use simple tools that allow user to define the contract for a service. This becomes part of the automated build before code for the new service is even written. The service is then built out only to the point where it satisfies the contract - an elegant approach to avoid the 'YAGNI'[[9]](http://martinfowler.com/articles/microservices.html#footnote-YAGNI) dilemma when building new software. |
| Decentralized Data Management | Decentralized data storage implies that each Microservice may have it’s own database (or may share it with some other Microservice). | Visualize data and system structure, show references between databases | Support patterns – different databases, |
| Infrastructure Automation | Infrastructure automation means continuous testing, building and deploying in different environments. The term is also called continuous delivery. |  | Provide integration with continuous delivery tools(Jenkins, TravicCI, DockerCI) |
| Design for failure | it is important to understand how service failures cause user experience  - system constant monitoring, checking the system metrics (number of queries) and any business metrics ( number of orders, downloaded files, seen pages).  Besides, the system should be ready for failures and the team should be ably to react as fast as possible | Provide convenient monitoring dashboards, which reflect the statuses, messaging systems | Provide monitor tools for user and system activity (such as Travic Ci, Code Cov). |

* 1. **Patterns**
     1. API Gateway Pattern

In common case, client service may want to interact with all services in the system. In such a way, we need to support that interaction directly, which makes the system less scalable and more complex. Client would have to connect to all end-points of each service. Besides, each service may use its own protocol, which could be not browser friendly and client would have problems with it.

In order to avoid the mentioned problems, API gateway pattern may be used. API Gateway is a single entry point into the system, which provides an access to the system to all clients. Moreover, API Gateway may serve as a controlling unit, which performs authentication, load balancing, monitoring, request shaping, caching and static response handling (Figure 8).

F:\Курсовая магистратура\картинки\API Gateway.png

Figure 8.API Gateway.

As far as API Gateway encapsulates all services in the system and the number of the services may infinitely grow, API Gateway must be scalable. In order to decrease the complexity of API Gateway implementation, special libraries may be used, for instance, Spring Reactor. Spring reactor is based on the well-known Reactor Pattern, which is an implementation technique for Event-driven Architecture. [12]

Traditional asynchronous callback approach makes the application tangled, complex and vulnerable to the errors. In order to avoid these problems, we may use reactive approach. The most famous implementation is ReactiveX, which is supported by many platforms. [13] ReactiveX extends Observer pattern and additionally allows declaratively compose events sequences on the high level of abstraction. Reactive Observables model abstracts from complicated callbacks webs and allows to treat the event streams as easily as data streams.

Another responsibility of the API Gateway services is errors handling. Without controlling the errors, the system may block, while the functionality of the broken service may not be so crucial for the whole system. In order to avoid blocking, circuit breaker pattern should be implemented. [14]

Circuit breaker pattern helps stops cascading errors across services, monitors the system, and provides fallback functions and increase the resiliency of the system.

The idea of the Circuit breaker is the following – in the circuit breaker we add protected function call which monitors for failures. When an error occurs and reaches the circuits breaker threshold, circuit breaker blocks the call of the function. Then, the circuit breaker monitors the state und keeps the function blocked until it recovers.

One of the efficient open source implementation of the circuit breaker pattern is Hystrix from Netflix.

* + 1. Load Balancer

Let’s consider Load Balancer, there are different approaches of balancing the resources. We will look at Apache Load balancer and Nginx (Table 2) [15].

Table 3.Load balancer policies.

| Policy | Description | Implementation |
| --- | --- | --- |
| Round-Robin | Distributes requests evenly according to the server weights. By default servers have weight =1. | Nginx, Apache, |
| Least-connected | The request is sent to the least loaded server in consideration with weights | Nginx |
| ip-hash(sticky) | Server is chosen according to the client IP address as it’s Hash function. | Nginx, Apache Camel |
| Hash(sticky) | Similarly to ip-hash, Server is defined by User characteristic, but could be any defined key | Nginx, Apache Camel |
| Least time | Preferable Server defined by smallest number of connections lowest and average latency, which is counted either by header (first byte receiving time) or last\_byte (the full response receiving time) | Nginx Plus |
| Random | The random server is chosen | Apache Camel |
| Topic | Requests are sent to all available servers | Apache Camel |
| Failover | If the error occurs on one end-point, it tries another end-point | Apache Camel |

* + 1. Microservices Inter-process communication patterns

Naturally, in Microservice architecture the issue of inter-process integration arises. Let’s consider possible approaches and existing solutions in the following field.

There are two types of the Inter-service communication:

* + - Asynchronous;
    - Synchronous.

The choice should be made depending on the system’s needs. Let us consider both types, provide most frequently used patterns and technologies.

Usually, the Microservices are Rest based because of the simplicity. The problem of using the pure Rest communication is that it makes the Services communicate synchronously and directly.

In **Synchronous communication**, the sender waits for the service response. The most obvious problem lays in the definition of the approach-sender is blocked while waiting to the response. The failures possibility should be treated very carefully.

By using Pipelines with REST protocol, we don’t need to support direct connection between services anymore (Figure 9).

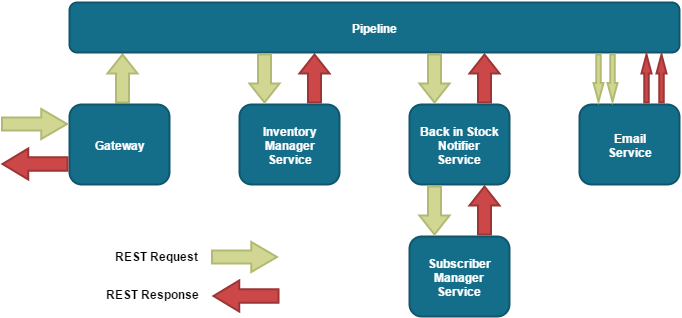


Figure 9. Pipeline.

The responsibility to organize messages is now on the pipeline. However, that makes services not self-sufficient, which is not a good practice for Microservices [16].

**Asynchronous Message-Based communication** is a well-known approach according to which the service places the messages into the channel, one or more subscribers receive the message. If the sender just needs to inform the subscribers, the work is done. If it needs the response from the subscriber, the sender listens for the channel, but does not blocks it’s work. There is a wide variety of open source messaging systems, the choice should be made due to the requirements and limitations of the services.

Usually, the services are not connected to each user directly, they follow event-driven pattern, so that the services are subscribed to required events. For the control purposes Message Broker pattern is used. Services place the message in the queue and continue their work while Message/Queue broker translates the message to the service (Figure 10).

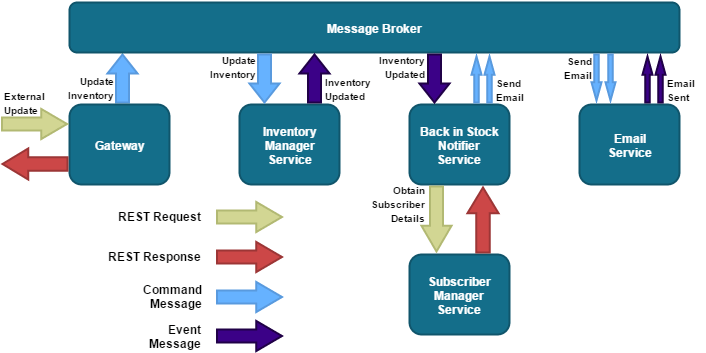


Figure 10.Message Broker.

In such a way, microservices become autonomous. The problem is a complexity of the Message broker, the flaw of the messages becomes unclear, the possibility of the error increases.

There are several implementations provided by different vendors. The most common implementations of the message broker are Apache Kafka, RabbitMQ, ActiveMQ and Kestrel .

Therefore, the Asynchronous approach with message broker best corresponds the highly scalable and loosely coupled nature of Microservices. Besides, asynchronous messaging with message broker may be implemented by event-driven architecture.

* + 1. Database patterns

According to the methodology, each Microservice should have it’s own database (Figure 10).

Services may also share different schemas of the single database or just different tables. The last two approaches make application less scalable, so the first approach is more preferable. However, the usage of different databases causes problems with data update. [17]

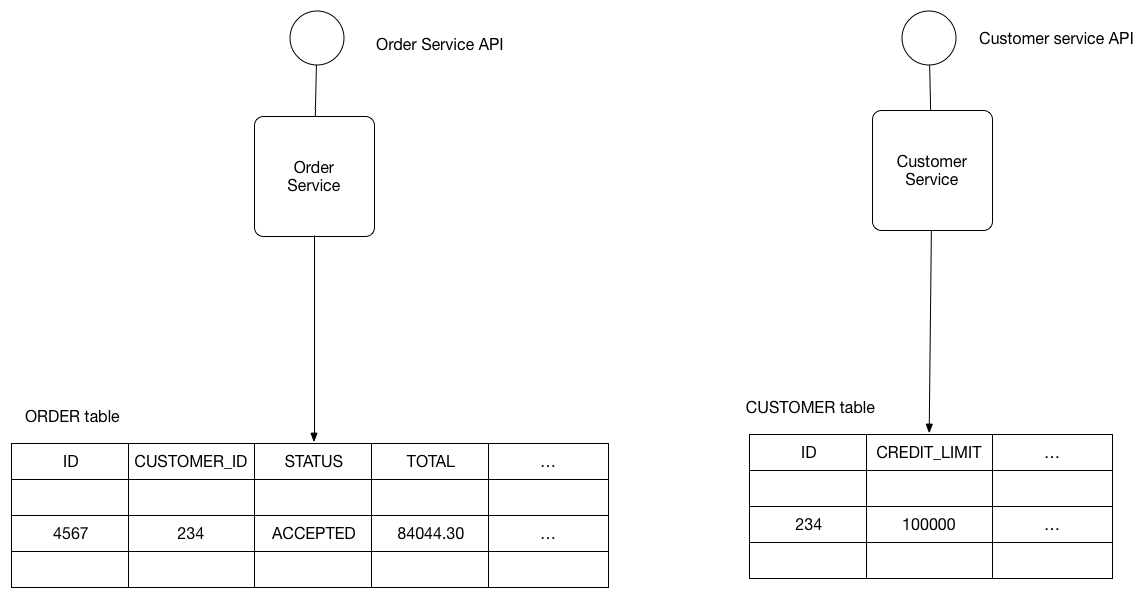


Figure 11.Separate databases architecture.

**Event-driven architectural pattern**

One of the solutions that helps to keep the data in different databases persistent is to use event-driven architecture. Each time the data is changed in one database, the service publishes an event (Figure 11). All services, which are subscribed on this event, update the data (Figure 12). [18]

D:\Курсовая магистратура\diagrams\database1.png

Figure 12. Publish event.

D:\Курсовая магистратура\diagrams\database2.png

Figure 13.Subscibe for event.

In order to achieve the atomicity of the database operations and updates we can use additional service, which will act as events’ queue, so only local transactions are performed. Each service has an access to the event table, where they place the events; Events’ message service takes the first event from the table and places it in the Message Broker.

Another approach is to use Transaction Log Miner instead of events table and events queue process. After the service changes database, it is reflected in the Database transaction log. Therefore, Transaction Log Miner may read the log and publish an event to the Message Broker.

D:\Курсовая магистратура\diagrams\database3.png

Figure 14.Transactions Log Miner.

Another well-known event driven approach is event sourcing which implies that the Event Store (events database) persists not the current state of the entity, but the sequence of the state-changing events. By replaying the events, it is possible to reconstruct the entities’ state (Figure 13). Event saving and publishing is an atomic operation, besides, all transactions represent itself as a well-structured log, which helps to easily track the operations. [19]

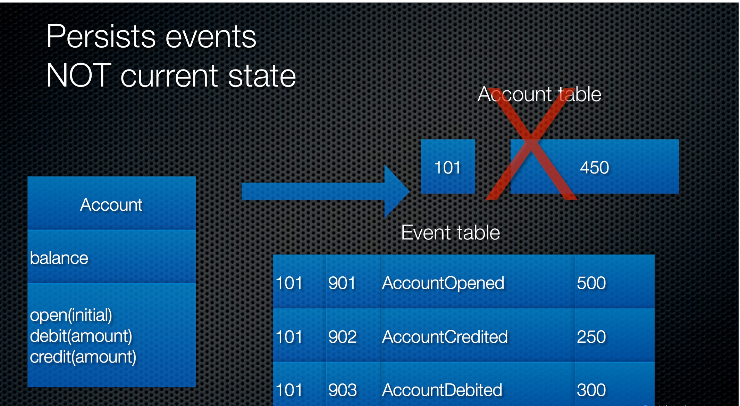


Figure 15. Persistent Events storage.

However, the pure event –sourcing is not sufficient, if join transactions of different entities are required. Let’s consider Command-Query-Responsibility-Separation pattern.

As it follows from the name, the pattern separates the queries (Read Model) and commands (Event Store) of the data (Figure 16).

D:\Курсовая магистратура\diagrams\Event_sourcing+CQRS.png

Figure 16.Command Query Responsibility Separation.

In compliance with Event Sourcing approach, the list of events is stored in Write DB. When the command side publishes events after updating or aggregating the view, the Query side consumes the event and recovers DB state, which is stored in Read DB.After that it returns the data view to he UI.

The separation increases the performance of the system and makes it more scalable. At the same time, the complexity of the system could be the reason for the errors, so the implementation should be very stable. [19]

* + 1. Service discovery pattern

The network locations of the service’s instances changes dynamically, thus, in order to refer to the service we need to use Service Discovery techniques such as Client-side discovery and Server-Side discovery.

In client-discovery client determines services instances network location and performs load balancing. That is performed via Service Registry.

In server-side discovery services request Load Balancer, which is implemented apart from the client. After that, Load Balancer queries Service Registry.

Service Registry is a database that stores service instances network locations. Besides, there are different approaches of services registration and deregistration. According to the **Self-Registration Pattern,** services themselves register and unregister the instances. The pattern is easy in implementation, but it requires implementing the registration function in each service. Another pattern is **The Third-Party Registration Pattern**, which implies usage of the so-called service registrar that tracks services activity, when there is a new service instance it registers it with Service Registry. The pattern suggests more difficult structure of the system, but allows avoid service registry functionality implementation on the client side.

* + 1. Patterns Comparison

In order to summarize the patterns, the following table is provided (Table 2.Patterns' Comparison).

Table 4.Patterns' Comparison

| Pattern | Description | Advantages | Disadvantages | Products |
| --- | --- | --- | --- | --- |
| Client-Service communication Patterns | | | | |
| Direct Client-Service Communication | Client Directly interacts with each service | No additional components required | Thick-Client,  Services may use different protocol,  Not user friendly protocols |  |
| API Gateway | Client interacts with API Gateway, which in turn interacts with all services | Client requests API Gateway, all work is on it.  Scalable  API Gateway may implement circuit Breaker and load balancer functions | The need to develop and maintain one more complex component |  |
| General Asynchronous approach (API Gateway) | Simple asynchronous messaging between API Gateway and Services | Simple implementation | -Tangled, complex and errors vulnerable  -Complex Queries may imply the requests to different Services, which may lead to many errors | Spring Reactor |
| Reactive (API Gateway) | Use events to communicate with Services | -Events simplify the information transferring  -Products usage allows to abstract from low level implementation  -Events flows may be treated as easily as data flows |  | Future in Scala, CompletableFuture in Java 8,  Promise in JavaScript,  ReactiveX, RxJava |
| Circuit Breaker | Function is wrapped in the proxy that is connected to the circuit breaker, so that it can be monitored. When the errors occurs and reaches the threshold, the circuit returns and error but does not call the function | -Helps to stop cascading errors  -Allows continuing the work  -Rapidly informs about failures place |  | Spring Cloud Netflix Hystrix, |
| Load balancer | Helps to maximize throughput, optimize resource utilization, reduce latency, and ensure fault-tolerant configurations. |  |  | [Camel Load Balancer](https://camel.apache.org/load-balancer.html) |
| Inter-processes patterns | | | | |
| Asynchronous Message-Based communication | Service sends the messages and continues the work; receiver, subscribed for this message, receives the messages and returns the receipt to the sender. | Supports independent and loosely interconnected services | Complicated Implementation |  |
| Synchronous Message-Based communication | Service sends the messages and blocks if it needs the response, receiver, subscribed for this message, receives the messages and sends the response to the sender. | Easier implementation | Service have to block while waiting for response |  |
| Asynchronous Message Broker/Pipeline | The Sender Service puts the message into the broker , the broker transfers the message to the Receiver Service | The Services are autonomous, no blockings | Complex implementation of the broker | Apache Kafka, RabbitMQ, ActiveMQ and Kestrel |
| Database Patterns  Eureka service registry | | | | |
| Single database, separate tables or schemas | Services share the database, but use different schemas or tables. | Easier way of data update, the data is always coherent | less scalable, Services are less independent |  |
| Different databases | Each Service has it’s own database | Services are scalable and independent | The data update is complicated, because an update should be the autonomous operation |  |
| Event-driven architectural pattern (Different Database) | Each time there is a change in database, service places an event to the message broker. Subscribers update the data and may publish new changes | The consistent data, the possibility of the materialized views | More Complex, errors should be gently treated, duplicate evens are possible |  |
| Publishing events to the local Event table | Service changes data and places an event in the local event table, Event Service takes the event, mark as completed and publishes it to the message broker | Local transactions, no need in two-phase commit | Error-prone, challenging NoSQL databases |  |
| Transaction Log Mining | Database transaction log represent the database changes; Transaction Log Miner Service tracks the log and publishes events to the Message Broker. | no need in two-phase commit, Service just updates the database | Transaction logs may differ in different databases and versions,  It may be difficult to recover an event from the low-level log | LinkedIn Databus [20]  DynamoDB [21] |
| Event-sourcing | The sequence of the state-changing events is store. | The data is consistent, changes are always published, stores the log of the changes | Insufficient for joint operations |  |
| Event Sourcing+CQRS | The changes are received by the Event Store and stored in Write Database. Then the event is sent to Read Model, which recovers the DB state form Read database. | Good performance and scalability, the data is consistent. | Complex, the implementation should be very stable. |  |
| Self-Registration Pattern | Service registers itself and update the table | Easy implementation | Each Service should have registration function | Netflix OSS Eureka client, Spring Cloud with Eureka, |
| Third-Party Registration Pattern | Registrar service tracks services activity , registers and unregisters them | Unique registration function in one service | One more component, more complex structure | NetflixOSS Prana |

# Existing EHS enterprise Web Application solution

The system, which was analyzed, refactored and migrated in the current work, is called Emex and it is developed and maintained by of the same name Irish company called Emex [22]. For the purposes of the intellectual property safety no business ideas, concrete technical solutions or schemes are provided.

The initial solution of the application has a single database, web application and several serving services. Web application consists of two main parts- server (which provides an API) (2) and client, which is a Single-Page application. The Entity Model service exploits Entity Framework functionality for retrieving the data from the database. The architecture of the initial solution is illustrated in the following diagram (Figure 17):

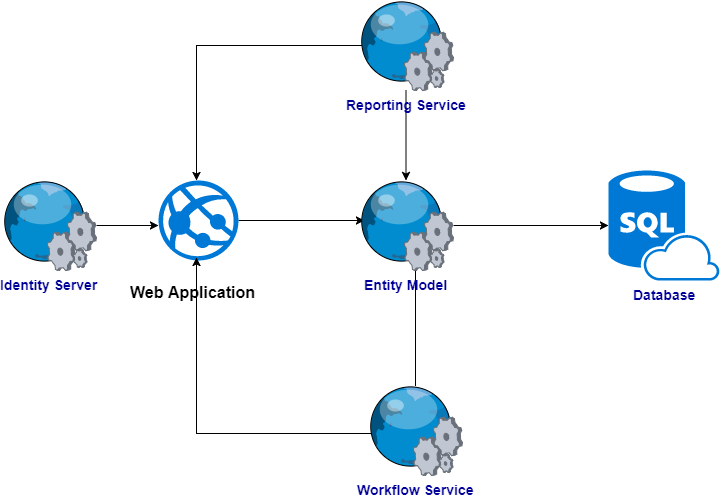


Figure 17. Initial solution Architecture.

* 1. **Business Scope**

The business scope is not in the focus of the current work, but in order to form the understanding of the meaning of the implemented processes it is necessary to provide several clarifications.

The most common user story for using the business part of the system consists of the following steps:

1. Log in to the system;
2. Navigate to the required module;
3. Find the record from the list of records;
4. Create a new record/ update the record;
5. Update the status of the record when it is required according to the business process.

The most common user story for using the admin part of the system consists of the following steps.

1. Log in to the system;
2. Navigate to the admin module;
3. Configure the system settings.

The workflow services are responsible for changing the state of the module record according to the business process lifecycle and send required notifications about the proceeded and required actions.

The system consists of EHS modules, which allow to simulate the steps of the client business process and to manage the data. Apart from EHS modules, the system contains Administration module and Organizational modules. Administration part is responsible for system configuration, by the means of administration module the customized client solution may be built without the development. In the current work, we refer to the System builder as to the client system, which could be configured by the means of the administration part.

All modules can communicate through the database. Most often modules share Organization information (such as employees, business units and business structure). Admin module does not influence the business data; it mostly refers the models. In the picture below the relationships between the modules are schematically illustrated. The connections are logical and are supported through the single database (Figure 18).

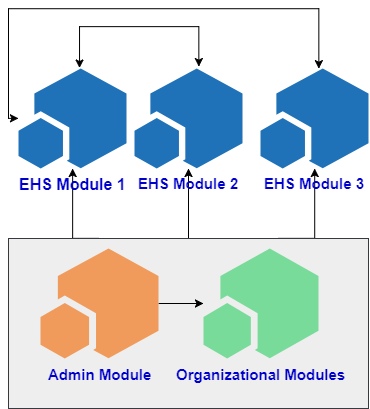


Figure 18.Logical domain schema

* 1. **Advantages of the current solution**

Advantages of the system lie both in the scope of clients’ interests and the development efficiency. From the client’s point of view, the system is meant to provide a rich functionality for the customization. The customization property is provided by the database logic, which is the most valuable part of the system. The database stores the settings for structural parts of the system like the fields in User Interface (UI (3)), the assessment models in Risk module and many other settings.

From the developers point of view the stack of technologies is significantly efficient, the Single-Page application (SPA) approach decreases the amount of requests and safes the resources. In comparison with the previous version of the system, which was an MVC project, the system performance became more efficient.

Namely, the application is using .Net Framework 4.2, AngularJS 2 + Typescript 1.7.Grunt is used for building the angular. Visual Studio 2015 is used as a development environment. TeamCity is used as a CI tool.

Overall, the system concept and expected functionality, in addition to the narrowly specified domain area, is of great interest.

* 1. **Disadvantages of the current solution**

There are many other advantages of the system, but from the maintenance point of view it is necessary to concentrate on the disadvantages. Essentially, in the process of system refactoring the existing advantages should not be lost.

One of the considerable drawbacks, which was noticed even by the clients, is a high response time. Despite a new architectural solution (new in comparison with MVC) was claimed to increase the performance, it did not reduce all the problems. The reason for such evidence consist in the fact that the system stores API and Client in one project. All requests to the system are being made and asynchronously processed by a single web service. Namely, the Web Application represents itself a rich client.

Another problem is a poor system scalability. Although the system is meant to provide a customizable client solution, which essentially implies many change requests to the system functionality, currently the global changes to the system itself are very cost effective. This is a common problem of any monolithic application. It is claimed that since the system is considered to be a final off-the-shelf solution, the inner architecture of the system should not be changed under the client requests. However, the last version of the EHS system, which is under consideration in the current work, is not a final off-the-shelf solution, it constantly requires refactoring and reorganization. Therefore, a new vision of the architecture is fully in the scope of the product development plan. Besides, one more weakness of the system consist in existence of the multiple versions of the solution – for each client. This makes the system maintenance very complicated and expensive. Versioning of the API could have solved the problem.

One more negative consequence, which is derived from the union of the API and the client into one web service, is inability to reuse an API by other client side applications. For instance, mobile application has to either use it’s own API, or call the API of the Web Application, which implies even bigger load on the server and in general is not a consequent decision.

A single database solution has both it’s own advantages and disadvantages. Essentially, the tables and other database entities are considerably interconnected. Therefore, the task of the database division will be one of the most sophisticated. One of the possible reasons for database division, as well as API division, is a fact that all clients request different modules of the system. Currently the system provides around 10 modules, if the client actually needs only three, there is no sense in delivering all system functionality. Providing the database schema, for which the client did not pay, contradicts intellectual property conventions. Now, the modules for which the client did not pay, are just being switched off for the client, while ideally the unnecessary parts should not be included into the provided solution at all. The analysis of the database will be provided in a microservices solution section.вообще нет)) мы же не раскрываем базу

Currently the system is being installed at the client servers. The development of the last version of the product was mainly conducted under the client specific requests and in a very short time, therefore the easiest way was to decouple each client version into a separate product solution. Further, the process of different versions unification was long and complicated. First of all, the simple versioning of the Microservices API will definitely solve the problem. Second, for small companies, which are not interested in buying the servers, the concept of Platform-as-Service may be implemented, which will be described in section 7 .

* 1. **Current solution performance assessment**

The official requirements to the existing solution are provided in the table below (Table 5).

Table 5.Official technical requirements

| Number of concurrent users | Recommended bandwidth | Minimum throughput |
| --- | --- | --- |
| 5 | 180 kb / s | 100 kb / s |
| 25 | 1 mb / s | 500 kb / s |
| 100 | 4 mb / s | 2 mb / s |
| 400 | 16 mb / s | 8 mb / s |

# Microservices solution for EHS enterprise Web Application

**TODO**

* 1. **The motivation**

Existing problems, described in the section 3.3, invite the architect of the system to consider a new approach. In the current work, we make an assumption, that Microservices architecture will increase the performance of the system and improve the scalability. Besides, the commonly applied in the scope of Microservices tools and techniques (Docker, Clouds, CI (5) and DevOps (6)), which in fact could be also applied without changing the architecture of the solution, but which are absolutely necessary when implementing Microservices, are expected to significantly simplify and accelerate the work process on each lifecycle phase of the system development, deployment and maintenance.

As mentioned before, Microservices approach is not radically new. The idea of providing multiple services for implementing specific parts of the system functionality is the basis of any Web Services application. Currently considered EHS solution is already using some web services such as Workflow service, Reporting Service and some others. Therefore, some functionality is already decoupled and subsequently less efforts should be applied for division the system functionality into independent Microservices. Therefore, the concept of different services answering for the specific scope of work is reasonably essential for the considered system.

However, the solution still has a single database, which makes the system parts interdependent. As it was mentioned above, the system consists of the EHS modules, admin part and a few organizational modules. Most of the time, EHS modules do not share the information, so that they are generally independent. The module database tables and other module related entities may be separated into the single database.

To summarize, the following arguments could be considered as the motivation for choosing Microservices architecture:

1. Loosely coupled domain logic;
2. Already existing decoupled web services;
3. Requirement for scalability.
   1. **Domain logic division**

The logic of the system implies the division into the modules. An essential division of the domain logic is the best possible solution for Microservices. Therefore, we will separate the module database entities into the separate database. Some modules may be very tightly coupled, so that it would make sense to keep them in one Microservices, either on the level of database, or by means of additional API Gateway.

In the picture below the part of the database is illustrated (Figure 19). The clusters of tables are the valid candidates for a separate Microservice. In the current work we will not implement the whole functionality of the system, but we will provide the prototype for the Equipment module and Admin module, which will be a separate Microservices. All other Microservices may be implemented similarly on the base of the existing prototype. Besides, as it was mentioned above, we need to consider the structural modules, such as People or Group. Almost all modules refers this data, which is in fact not very often updates - the changes in the organizational structure of the system are not regular, as well as People. Therefore, we will duplicate the required organizational tables into the database and we will subscribe the Microservices to the subsequent update events. The choice of Microservices intercommunication pattern will be described in the section below.

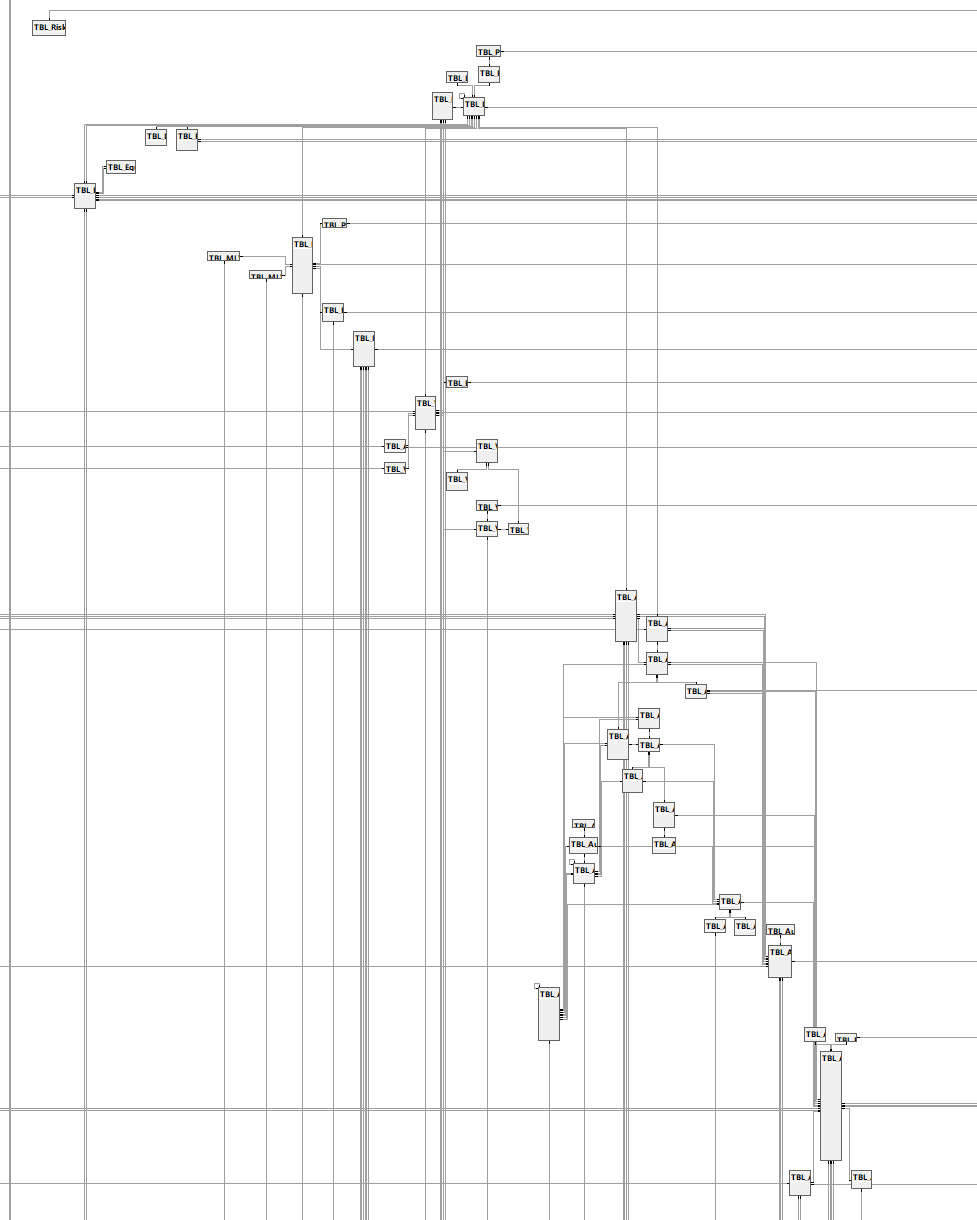


Figure 19.Current Database Schema

# Client – Server Communication

In the current version of the system the web application is a rich client, which has both client and server functionality. When applying Microservices approach, we want to decrease the load on each functional module, so first we separate the server from the client and then separate all the API’s into separate Microservices. The most common operation for each module is a CRUD operation, we do not expect the clients to manage the most of business data simultaneously, hence, simple asynchronous API Gateway pattern would be applicable for implementing Client–Server Communication. In the API Gateway load balancer and Service discovery services could be implemented. In the scope of the current work we do not implement this functionality. In the picture below a schema of the Client-Server communication is provided (Figure 20).

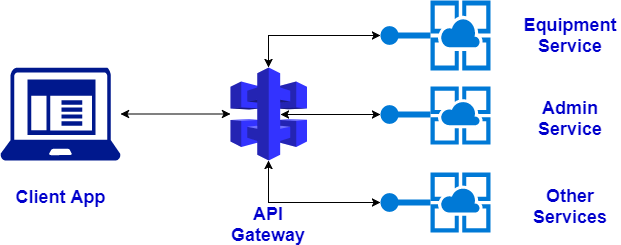


Figure 20.API Gateway

In the web application, each module has a set of the main components, which implement relatively the same functions, but for each specific module. Besides, each module may have module specific components and directives, which are applied only in one module.

AngularJS has a reach functionality for building the component-based architecture, while component itself is essentially a key part of it. Component-based architecture of the Web application allows reaching a high-level scalability and performance.

The communication with the server from the client side is invoked by the main module component, which in turn calls the required service of the web application. In AngularJS services are substitutable objects that are wired together using dependency injection (DI) and that can be used to organize and share code across the application and outwards the application [23]. Namely, for requesting the data for Equipment module, Equipment component calls the Equipment Service, which uses HTTPClient for performing the Get Request. HttpClient is an injectable class for performing http requests. The Equipment module subscribed for the result of getEquipmentItems() method of EquipmentService (Figure 21). As soon as the result will be received- the data of the Equipment Component will be updated.

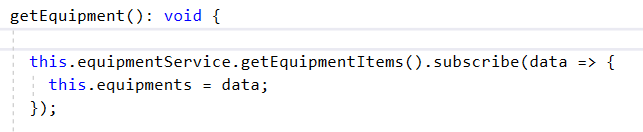


Figure 21

EquipmentService, as well as the whole application, does not know how to reach Equipment API. All modules’ APIs are decoupled from the Web Application. But the EquipmentService can send the requests to API Gateway, which in turn either sends the notification to all subscribers for updating/deleting the data, or request the data from the subsequent modules. Therefore, for getting the Equipment data, the Equipment Service sends get request to API Gateway, which redirects the requests to Equipment API. Equipment API exploits an Entity Framework for building the model by using Database First approach [24] . Due to the small database (only one module), the Entity Model is very light and fast. Equipment Server retrieves the data from the database by means of the model, returns it to the API Gateway, which in turn delivers it to the Web Application.

# Inter-process communication

Communication between different modules reduce to reading or updating the data of the other services. For instance, the records from the Equipment module may be connected to Audit module. In case of big modules, such as Audit or Incident, we could not duplicate all the connected data from other modules- it well be not efficient. Hence, the Microservices need to retrieve the data from other Microservices databases. For these purposes, we simply exploit Get requests and API Gateway functionality.

*TODO*

Update and Delete requests will be more complicated, as far as multiple Microservices may contain updated data. We perform pretty simple operations and the data is not sensitive – there are no accounting or financial operations, therefore, there is no requirement for tracking the changes of the data ( usually this requirement concerns only the actions in the Workflow Service, as far is the clients are interested on the path of their business process). Consequently, for this specific system Event Driven Architecture will be optimal. Simple asynchronous update/delete operation will be too indiscriminate.

TODO

# API Versioning and SaaS

Versioning is one of the commonly used features of the Microservices API. As it was mentioned in the current solution analysis section, the old version of the system does not support API versioning. Meanwhile, such simple modification of the API will considerably increase the functionality and scalability of the system.

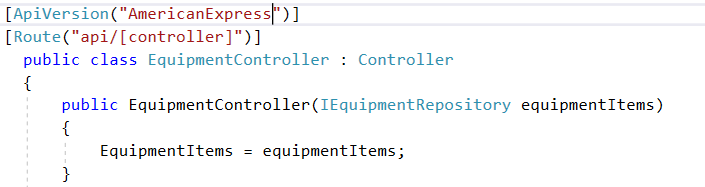


Figure 22.API Versioning

Versioning in case of the unique product may be used for tagging the client specific functionality. Besides, it would noticeably simplify the future refactoring of the system, as far as even while the old version of the client application will be referring the old API Version, we can still work on and test the new version.

In the scope of the API versioning, it is necessary to mention a system use case as System –as – a – Service (SaaS) (9), which is cloud-based application, which provides a full functionality of the system through Internet. For the clients with a small number of users buying the servers for the EHS system may be too expensive and excessive. They may need just one or two simple modules. In this case, the concept of SaaS is required. The picture below represents the notion (Figure 23) [25].

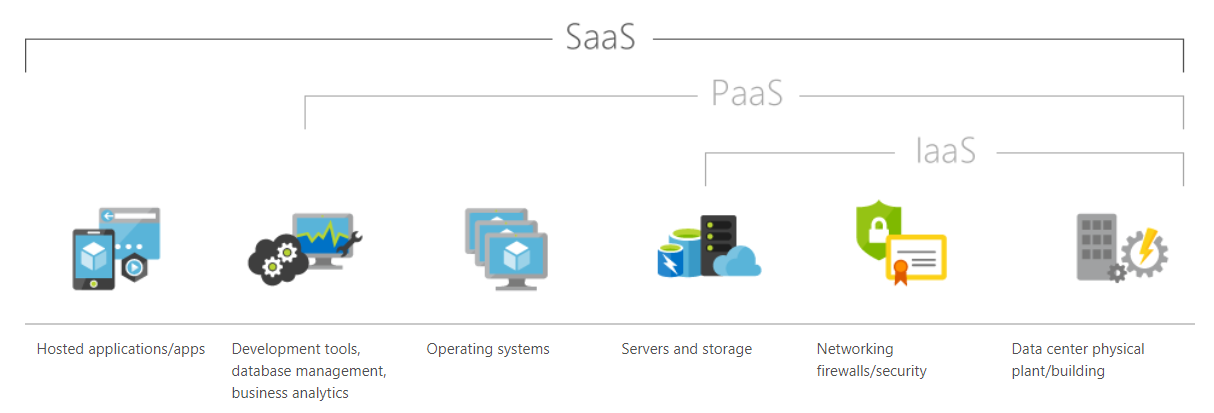


Figure 23.SAAS

In the scope of the SaaS approach versioning may be helpful for decoupling the databases or as well the API. In turn, the usage of Docker, which is essential for Microservices, will significantly simplify the migration to SaaS format.

# Dev-Ops

Due to the fact that during the last two years the popularity of the Microservices architecture was constantly growing, many companies and services provided additional functionality for Microservices implementation. Many of them imply accompanying Dev-Ops (2.2.3) activities.

Microsoft provided the guide for developing and maintaining Microservices [23], which is dedicated to the presentation of the new facilities, provided in Visual Studio 2017. Namely, currently Visual Studio supports integration with Docker. Besides, there is a description of Microsoft tools for Microservices, as API Gateway or Azure services as Azure Service Fabric and many others.

Therefore, building and deploying the system into the Docker and further testing of the systems now may be essentially accomplished by means of Visual Studio 2017. Therefore, the project was migrated to the 2017 version of the development environment. Now, the building and deployment could be easily done without any expertise in Team City.

# Results

Comparison of the old monolithic system and the new Microservices system is provided in the table below (Table 6):

Table 7.Old and New System Comparison

|  |  |  |
| --- | --- | --- |
| Property | Old System | New System |
| Scalability | Required the changes in web application and the deployment of the whole system | The changes in specific part of the system, work |
| Performance | Is good for a small amount of users,  Is low on the enterprise level. | The performance has assumingly slightly decreased for the small amount of users, but improved for the big number of users |
| Technologies novelty | * Angular 2 (5 version was officially announced this ) * Grunt – an old tool * TeamCity - requires additional configuration and expertise * Typescript v 1.7 | * Angular is upgraded to 4 – with only a slight changes could be upgraded to even faster Angular 5 * WebPack – much faster than grunt * Docker - provides a great functionality for system test and deploy * Typescript v 2.4 |
| Database scalability | Updates of the existing database,  Only MS SQL is possible | Updates of existing small databases or creation of the new database if the new module is added  For the new modules any database may be used |
|  |  |  |

TODO

The provided solution was accepted by Emex company and will be used in the development of the new v9 version of the system, which is proved by the certificate of implementation (Reference).

# Future development

The solution developed in the scope of the current master work is a structural prototype of the system, so only a part of the functionality is present. The most valuable aspect, apart from theoretical analysis and conclusions of the system itself is an architecture solution of the new Web application and servers. The prototype will be further used for the development of the system on the enterprise level.

In future we would like to decouple the web application more- each module contains module specific directives, sometimes the components are so big, that they represent itself a big part of the system functionality. The idea lays in the scope of dynamic injection of the components, which structure will be requested not from the client, but from the module server API. The idea is fundamental and the advantages and disadvantages of it should be very carefully analyzed.

Platform as Service and multitennatn application

# Summary

In the last two years Microservices have attracted a lot attention. What makes Microservices methodology very attractive is that the approach is in fact the description of the best practices, which were successfully applied by major companies. Although all began from the internet-user oriented web applications, now enterprises also understand the efficiency and the benefits of the clouds while modern technologies are able to provide required security level.

For many reasons monolithic applications no longer meet the requirements of the complex enterprise systems. Despite the fact that well-known SOA provided many brilliant ideas and conceptions, it has not become very popular and widely used, because of the complexity and high abstraction level, however, in some way SOA has laid the foundation for Microservices architecture.

The application of the Microservices methodology is a very challenging process for many reasons. The concept of division the business scope and subsequent division of the functionality and responsibly makes the process inconvenient and unfamiliar for the team. Besides, apart from Microservice development, the team should be familiar with continuous integration and continuous delivery concepts. Considering that the system may contain over 10 services, and subsequently, 10 teams and 10 databases, the project and system management becomes complicated and messy.

After conducting the analysis of the Microservices characteristics, the solution for the new architecture of EHS enterprise system was developed and prototype was implemented. Microservices related patterns were analyzed and the most appropriate were included into the final solution. The final part of the work consists of the architectural solution and system prototype.

# Conclusion

The goal of the current work was to analyses Microservices architecture, analyze existing EHS solution, asses the profit of migration of the old system to the Microservices, in case of the positive result develop the architecture, and implement the prototype. As a result, it was concluded that Microservices will be very effective for (this) kind of system.

# Acronyms and Terms

1. EHS - Environment, Health and Safety
2. SOA - Service Oriented Architecture
3. API - application programming interface
4. UI - User Interface
5. SPA - single page application
6. CI - Continuous Integration
7. DevOps - development и operations
8. CRUD - Create Read Update Delete
9. SaaS – System-as-a-Service

# Bibliography

|  |  |
| --- | --- |
| [1] | R. Stephens, Beginning Software Engineering, John Wiley & Sons. |
| [2] | M. Fowler, "Microservices," 25 March 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed 07 11 2017]. |
| [3] | MSDN Microsoft, "Chapter 1: Service Oriented Architecture (SOA)," MSDN, [Online]. Available: https://msdn.microsoft.com/en-us/library/bb833022.aspx. |
| [4] | I. Dwyer, "MICROSERVICES: Patterns for Building Modern Applications," 2015. [Online]. Available: https://www.iron.io/resources/white-papers/microservices-patterns-for-building-modern-applications/. |
| [5] | J. Rivera, "Gartner Identifies the Top 10 Strategic Technology Trends for 2015," Gartner, [Online]. Available: http://www.gartner.com/newsroom/id/2867917. |
| [6] | M. Fowler, "Microservices a definition of this new architectural term," [Online]. Available: http://martinfowler.com/articles/microservices.html. |
| [7] | Dr. Paul Chapman, "Microservices with Spring," Spring, 14 07 2015. [Online]. Available: https://spring.io/blog/2015/07/14/microservices-with-spring. |
| [8] | E. K. Barry Brigg, "Enterprise Cloud Strategy," [Online]. Available: microsoftpressstore.com. |
| [9] | B. Root, "The difference between Continuous Integration and Continuous Delivery," [Online]. Available: http://blog.nwcadence.com/continuousintegration-continuousdelivery/. |
| [10] | Jenkins, [Online]. Available: https://jenkins.io. |
| [11] | M. Hering, "How to Define Your Devops Roadmap," *The DZone Guid to Cintinious Delivery,* vol. 3, pp. 12-13. |
| [12] | JON BRISBIN, "Reactor - a foundation for asynchronous applications on the JVM," Spring, 13 may 2013. [Online]. Available: http://spring.io/blog/2013/05/13/reactor-a-foundation-for-asynchronous-applications-on-the-jvm. |
| [13] | ReactiveX, [Online]. Available: http://reactivex.io. |
| [14] | Chris Richardson, "Building Microservices: Using an API Gateway," 05 june 2015. [Online]. Available: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/. |
| [15] | Nginx, "NGINX LOAD BALANCING – HTTP LOAD BALANCER," [Online]. Available: https://www.nginx.com/resources/admin-guide/load-balancer/. |
| [16] | Craig Williams, "IS REST BEST IN A MICROSERVICES ARCHITECTURE?," Capgeminin, 18 December 2015. [Online]. Available: http://capgemini.github.io/architecture/is-rest-best-microservices/. |
| [17] | C. Richardson, "Pattern: Database per service," [Online]. Available: http://microservices.io/patterns/data/database-per-service.html. |
| [18] | C. Richardson, "Pattern: Event-driven architecture," [Online]. Available: http://microservices.io/patterns/data/event-driven-architecture.html. |
| [19] | C. Richardson, "Introduction to Event Sourcing and Command Query Responsibility Separation (CQRS)," [Online]. Available: https://github.com/cer/event-sourcing-examples/wiki/WhyEventSourcing. |
| [20] | C. Botev, "Source-agnostic distributed change data capture system," 15 November 2015. [Online]. Available: https://github.com/linkedin/databus. |
| [21] | "Capturing Table Activity with DynamoDB Streams," Amazon, [Online]. Available: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html. |
| [22] | "Emex," [Online]. Available: http://emex.com. |
| [23] | Microsoft, "Free ebook: Containerized Docker Application Lifecycle with Microsoft Platform and Tools," 2017. [Online]. Available: https://blogs.msdn.microsoft.com/microsoft\_press/2016/12/07/free-ebook-containerized-docker-applications-lifecycle-with-microsoft-tools-and-platform/. |
| [24] | Wikipedia, "Monolithic system," [Online]. Available: https://en.wikipedia.org/wiki/Monolithic\_system. |