Decomposition of monolithic web application to microservices

Mikulas Zaymus

Bachelor’s thesis
May 2017
School of Technology, Communication and Transport
Degree Programme in Information and Communications Technology

jamk.fi

Description

Author(s) Zaymus, Mikulas	Type of publication Bachelor’s thesis	Date May 2017
	Number of pages 56	Language of publication: English
		Permission for web publication: yes

Title of publication
Decomposition of monolithic web application to microservices

Degree programme
Information and Communications Technology

Supervisor(s)
Salmikangas, Esa; Huotari, Jouni

Assigned by
Solteq Oyj

Description

Solteq Oyj has an internal Wellbeing project for massage reservations. The task of this thesis was to transform the monolithic architecture of this application to microservices.

The thesis starts with a detailed comparison between microservices and monolithic application. It points out the benefits and disadvantages microservice architecture can bring to the project. Next, it describes the theory and possible strategies that can be used in the process of decomposition of an existing monolithic application to microservices.

The practical part of the thesis describes the implementation steps. It starts with the analysis of an existing application architecture and technology stack. After a detailed analysis of application the choices made in the process of design of decomposition to microservices are discussed.

The used platform for the new application design was Amazon Web Services. Among the used AWS services can be found: Amazon S3, Amazon Route 53, Amazon DynamoDB, Amazon EC2, AWS Lambda, Amazon API Gateway and Amazon CloudFormation.

The implementation was carried out by using Node.js and new microservices were buil based on AWS Lambda functions. For deployment and resource management of AWS Lambda functions Serverless framework was used.

The thesis results in the design for decomposition of a monolithic application to microservices and a partial implementation of the designed transformation process. The final solution was deployed and tested on Amazon Web Services.

Keywords (subjects)
Microservices, Cloud, AWS, AWS Lambda, Serverless framework

Miscellaneous

Contents

Figures

Acronyms

API	Application Programming Interface
AMI	Amazon Machine Image
ARN	Amazon Resource Name
AWS	Amazon Web Services
DDD	Domain-Driven Design
DNS	Domain Name System
EBS	Elastic Block Storage
EC2	Elastic Compute Cloud
EFS	Elastic File System
ELK	Elasticsearch - Logstash - Kibana
FaaS	Function as a Service
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IoT	Internet of Things
JSON	JavaScript Object Notation
MVC	Model – View - Controller
SaaS	Software as a Service
PaaS	Platform as a Service
RDBMS	Relational database management system
REST	Representational State Transfer
RPC	Remote Procedure Call
XML	eXtensible Markup Language

1 Introduction

1.1 Motivation

Information technology is a field of expertise that offers a wide variety of tools, frameworks and programming languages. In addition, all of them are instantly improved and modernized to keep up with demands of today’s customers and market. Choosing the right tools can help to accomplish a task with ease and bring value to both programmer and also a customer; however, on the other hand, by poor selection of technologies a company can lose its agility and render itself future technical debt. Technical debt may not be obvious at first sight; however, it may appear in the long run and cause stagnation of a project.

In the last decade significant shift of mindset and practices how web development is done can be seen. Traditional server-rendered applications built on MVC pattern backed by RDBMS are on decline. Modern JavaScript frameworks such as Angular and React are becoming more and more popular. Old relational database systems are being replaced by document-oriented NoSQL databases. These technologies bring new winds and challenges to the world of web applications. Among those challenges, there is the need for scalability, agility and faster delivery at an affordable price. Cloud computing overcomes many of these obstacles.

The cloud community grows each year hand in hand with number of successful business stories based on the cloud technologies. This fact confirms the last research revealing that AWS certificate is the most valuable of all IT certifications today (Columbus 2016). The cloud enables developers to leverage modern architectural designs such as microservices (Lewis, & Fowler 2014).

Microservices or microservice architecture is often beneficial for large enterprise applications that require high scalability. The hosting company, Solteq Oyj has long experience with managing IBM WebSphere Commerce, which is a good example of a monolithic application that could benefit from microservice approach. The purpose of this thesis is to explore what improvements and drawbacks come with the decomposition of a monolithic application to microservices (Fowler 2015).

1.2 Hosting company

Solteq Oyj is a medium-sized Finnish company providing software solutions for e-commerce. The company has 8 offices spread around 3 countries and employs around 500 people. During all years of operation, the main domain of expertise of company has been based on IBM WebSphere Commerce. In the last years, the company invests in learning of cloud technologies and tries to find its position in the market of companies offering solutions based on cloud services.

For purposes of learning, the company created an internal project based on the cloud technologies. This project helps new cloud developers to start up using cloud services and experiment with new ideas. This project was used in this thesis to try real application decomposition to microservices in a small scale.

1.3 Objectives

The main goal of this thesis was to learn about microservices as an architectural pattern. As a proof of good understanding of microservices and differences against monolithic approach, a strategy of architectural transformation was to be designed. This strategy should include a solution how to execute this transition and build microservices around a living monolith. Amazon web services were selected as a platform because it is currently the most mature and used cloud platform to this day (Panettieri 2017).

1.4 Thesis outline

This thesis consists of four main parts. The first part presents *Cloud computing* as an innovative way of hardware resource provisioning. This chapter also describes four main services provided by cloud vendors. The second part introduces *Microservices* and is followed by a deep analysis of pros and cons in comparison to monolith architecture. The third part of this paper, *How to decompose Monolith to Microservices* is dedicated to the theory behind the execution of application decomposition. It points out approaches that can be adapted during this process. The fourth part *Implementation* includes all aspects related to a real world example of decomposition to microservices including the design of strategy how to execute this transformation. The last chapter includes the conclusion of this thesis and reveals all findings of this thesis in a brief manner.

2 Cloud computing

2.1 History

In the process of software development, one of the key stages of successful software delivery is application deployment. To deploy a product, available physical machine is needed, which is powerful enough to run the application. In history, the customer's private data center usually provided this machine for deployment. The facility as well as the network had to be secured to prevent unauthorized access from outside. Such solution is also known as On-Premise. If the customer did not possess the required resources, the IT company could offer him their own resources. To maintain the company's own data center is not always profitable. It requires qualified employees and more additional costs in case of hardware failure. Another problem of in-house solutions is scalability, especially once capacity limits are hit and provision time of new resources is delayed by need of hardware upgrade, resulting in project delays. These and many more issues opened doors to a new way of tackling this problem: cloud computing. (Venkatraman 2011)

Cloud computing, often referred to as "the cloud" is internet-based computing, provided on demand. Cloud services are publicly available with "pay as you go" business model, which means one only pays for the used computational power. As a beginning of cloud computing year 2006 can be considered when Amazon introduced its Elastic Compute Cloud. Later on, in 2010 Microsoft released its Windows Azure, followed by IBM SmartCloud in 2011 and Google Compute Engine in 2013 (Cloud computing 2017). In the Figure 1 it can be seen that popularity of Cloud technologies is growing every year. It results in less money spending for traditional hardware and software infrastructure.

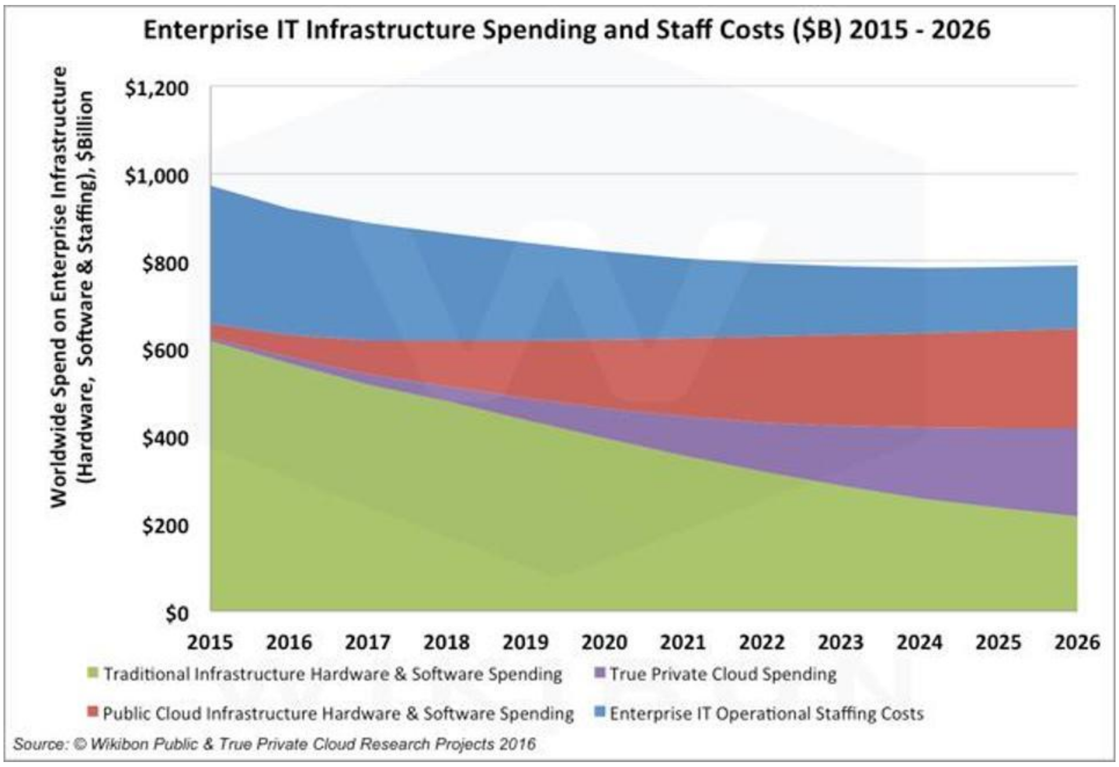

Figure 1. Forecast of the private Cloud revenue (adapted from Wikibon Public 2016)

Thanks to cloud computing, the developers can get rid of resource management burden and devote all their time to software development. Cloud computing comes in several service models (Cloud computing 2017):

- SaaS
- Paas
- IaaS
- FaaS

2.2 SaaS (Software as a Service)

In the Software as a Service, the user is granted access to a software application. Software installation and maintenance is provided by cloud providers. One of the disadvantages of SaaS is that all user data are stored outside of the user computer. To ensure the user’s safety providers allows users to use data encryption. This should increase overall safety, in case of data theft (Cloud computing 2017). Services such as Dropbox, OneDrive, Google Apps, and Microsoft Office 365 belong to SaaS (Vladimirskiy 2016).

2.3 PaaS (Platform as a Service)

In Platform as a Service, the user receives computing resources usually in the form of a virtual machine. The vendor usually provides a machine with a preinstalled operating system including a runtime environment or an application server. Such a server is ready for application deployment.

One of the main selling points of Platform as a Service is automatic scalability. This ability enables the provided platform to increase its resources according to the current load without any downtime. This is extremely beneficial for applications that need to handle an extreme load for a short amount of time on irregular basis. (Cloud computing 2017) A typical example could be software for trading stocks and shares.

2.4 IaaS (Infrastructure as a Service)

In Infrastructure as a Service, the vendor offers computing infrastructure. To this category falls file/object storage, firewalls, load balancers and others. They have an indispensable position in overall infrastructure of software solution. (Cloud computing 2017)

2.5 FaaS (Function as a Service)

Function as a Service is a fairly new concept, sometimes mentioned as Serverless. The key idea behind FaaS is to run code without provisioning or maintaining servers. The only requirement is to upload the code itself. With this also comes automatic scalability. The pricing model where one pays only for time of the code execution is very appealing, which can result in dramatic money savings in comparison with the same solution based on PaaS. (Cloud computing 2017)

3 Microservices

3.1 What are microservices

Microservices, sometimes mentioned as microservice architecture presents a different approach to application development. Rather than creating one single application, the microservice application consists of multiple small services, each running in its own process. Communication takes place via lightweight mechanisms such as HTTP, REST or some kind of RPC. Thanks to the fact that every microservice runs in its own process, each microservice can be deployed independently to others and a suitable programming language can be chosen for each of them respectively. (Lewis, & Fowler 2014)

Microservices come with many advantages in comparison with the monolithic architecture. In the following chapter, the main benefits and drawbacks of this approach are covered.

3.2 Microservices vs. Monolith

3.2.1 Small size

Microservices are often compared to UNIX command line utilities where each utility does only one thing and does it well. Similarly, one microservice should focus on a single unit of business domain and act in its context.

Every microservice should be developed by a smaller autonomous team. The size of the team may vary from case to case; however, a general unspoken rule claims that the amount of developers in one team should not exceed the number of people one can feed with two pizzas. One team is responsible for a full stack of technologies from UI and middleware to database administrators. This reduces time spent on communication between multiple separate teams divided by specialization. Teams can than operate in a more agile way and respond to business domain changes very quickly. The effect of communication reflected into design is also described by Conway’s Law (Figure 2 and Figure 3):

“Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.” (Conway 1968, 31)

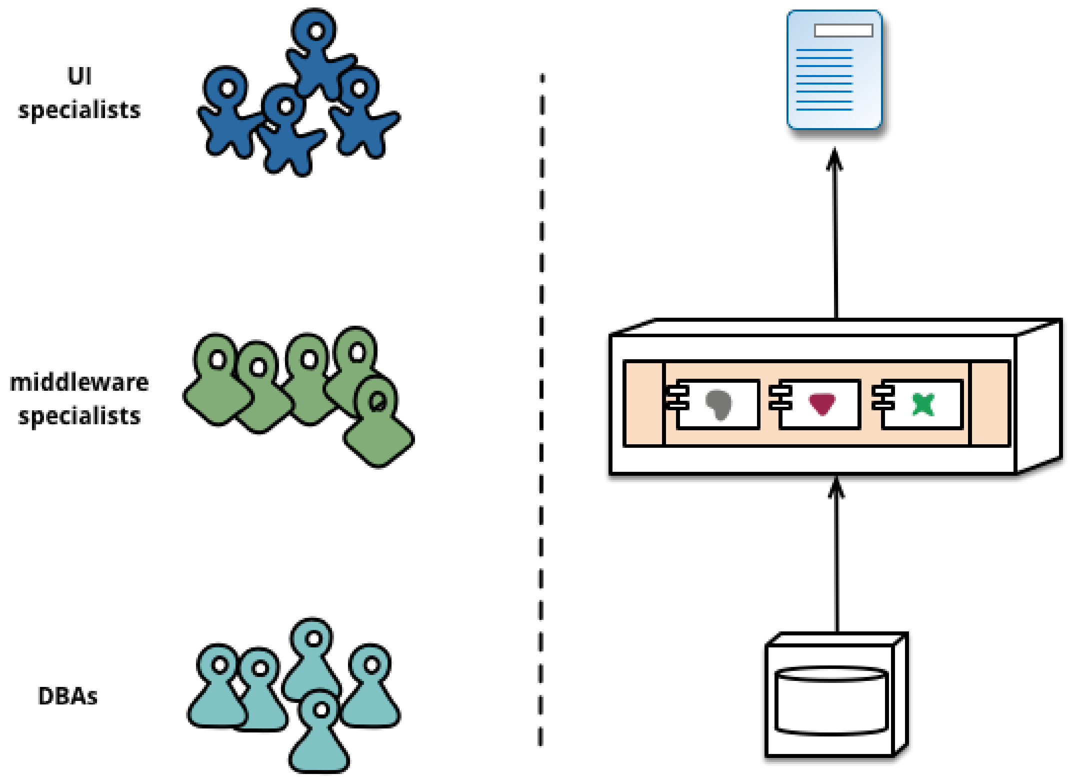

Figure 2. Conway's Law – Monolithic application (adapted from Lewis, & Fowler 2014)

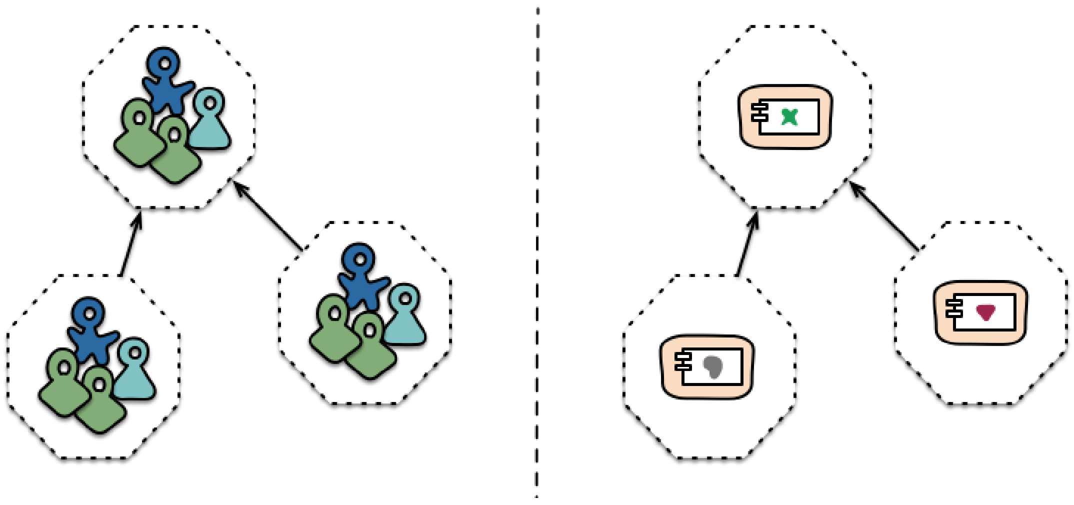

Figure 3. Conway's Law – Microservices (adapted from Lewis, & Fowler 2014)

Small size does not apply only to development team but also to the code base that the team manages. Small size of code enables the team to start from scratch and rewrite the entire microservice when needed (Goldsmith 2015).

New team members also benefit from smaller complexity of code base, and they can get on board really quickly thanks to a steep learning curve. (It can be stated that code is small enough to fit into one’s head.) The quick adaptation of new members may be leveraged to move people in between teams from a low-impact business area to higher-impact areas without high additional cost (Ranney 2016).

3.2.2 Technological neutrality

As mentioned in chapter *What are microservices*, each microservice runs in its own process. Thanks to this fact, developers are not stuck with one technology stack for the whole application. They can choose a different programming language, database system and storage solution for each microservice independently based on their needs. Difference between running application in a single process or separate processes is visualized in the Figure 4.

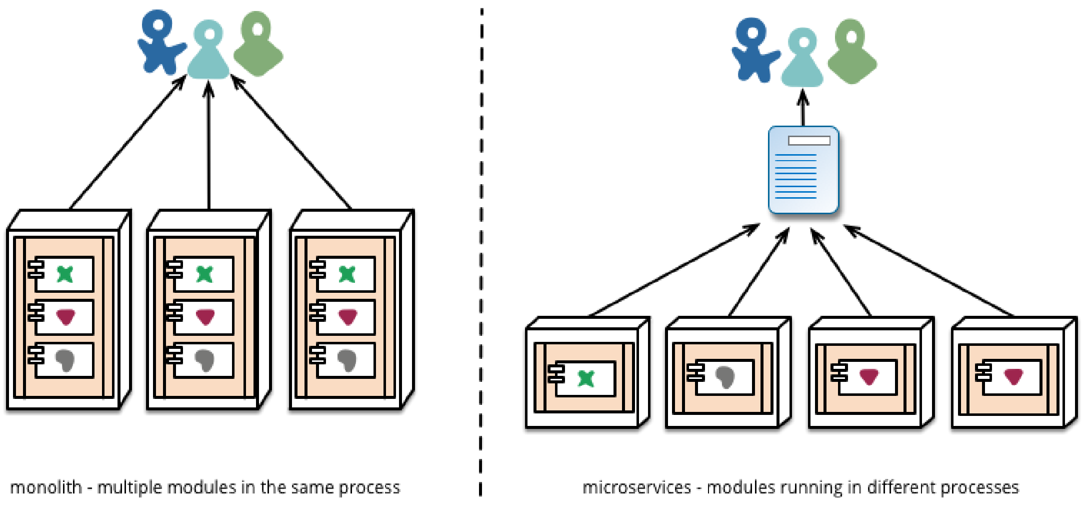
monolith - multiple modules in the same process
microservices - modules running in different processes

Figure 4. Each microservice runs in its own process (adapted from Lewis, & Fowler 2014)

However, using a wide range of programming languages should be avoided because it can lead to many problems, which are not so apparent at the first sight.

Matt Ranney (2016) warns about this problem in his talk at GOTO Conference in Chicago 2016. He says that using various languages tends to lead to a fragmentation of culture, where people are split to groups based on their preferred language. Changing people who use different technologies in between teams can cause difficulties.

Code sharing also starts to be less effective. Using a solution written in one language cannot be conveniently used in another team. Different languages usually do not feature the same monitoring tools, which can cause troubles in the process of profiling and searching for bugs in between microservices. An additional effort needs to be made to ensure that the output of monitoring tools is the same through different technologies. (Ranney 2016)

All these points taken into account, different technologies should be used sparingly. It can be more beneficial to stick to one programming language, unless there is good reason not to do so.

3.2.3 Scalability

When the application grows due to popularity and needs to handle more and more users simultaneously, running an instance can fast reach its limits. There are two possible ways how to overcome this problem. The first is to optimize the software to utilize hardware resources more effectively. This option is expensive and not always possible; thus, in the majority of the cases the second option is chosen: to increate hardware resources. (Ranney 2016)

Each physical machine has its limit and an endless increase of hardware resources such as CPU, physical memory, physical storage is not possible. When the system load is too big for a single machine, it is necessary to introduce a second machine with another running instance of the application. The load is then equally divided between the two machines thanks to a load balancer that has a role of the routing device (Anicas 2014).

By introducing a second machine system benefits not only from a higher maximum load but also from increased redundancy. In the case of failure of one machine, the second machine takes over the whole communication and the end user does not notice any downtime.

One of the main differences between monolithic application and microservices is that the monolithic application runs in a single process against multiple microservices, each running in its own process. For the monolith it means that the whole application has to fit into one machine because it is not possible to have one process running across multiple machines. That physically limits the single instance application to the maximum capacity of one machine. (Goldsmith 2015)

In the world of enterprise applications reaching the maximum capacity of physical machine is a real threat. That often leads the developers' team to change the strategy from monolith to microservices that are better designed for bigger scale (Goldsmith 2015).

Microservices run each in their own processes, thus they are not required to run all on one physical machine. This allows them to spread around multiple machines. If one of the microservices is used more frequently than others, it is sufficient to introduce only one microservice of that kind. It can be easily seen in Figure 5.

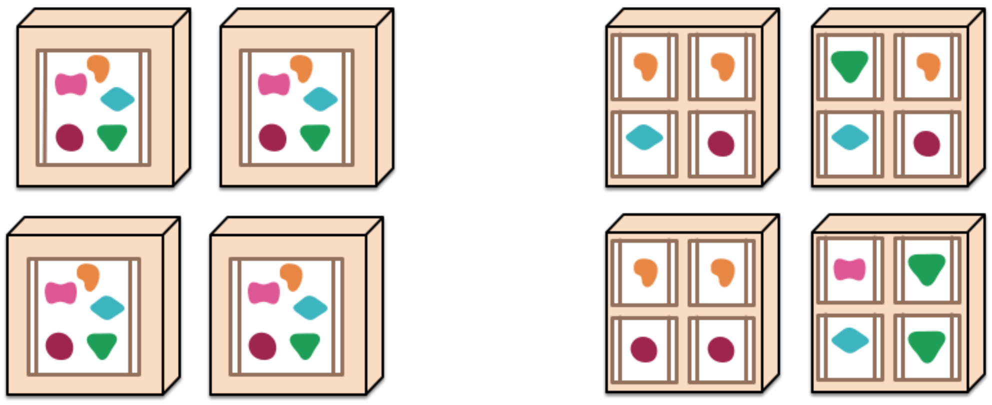

Figure 5. Scaling monolith vs. microservices (adapted from Lewis, & Fowler 2014)

This way of scaling can go further than the monolith and offers much more effective resource utilization resulting in money savings.

Martin Abott and Michael Fisher describe in the book The Art Of Scalability (Juneja 2016) how software can be scaled to reach its maximum scalability potential. They state that there are three ways how it can be done. The first of them (X-Axis) is by adding additional instances and this approach has already been covered in this chapter. The second way is functional decomposition (Y-Axis). This is the benefit that microservices bring to the table by their separation to multiple processes. The third and last is sharding or partitioning the database related to application or microservice. (Juneja 2016) All three scalability dimensions are displayed in the Figure 6.

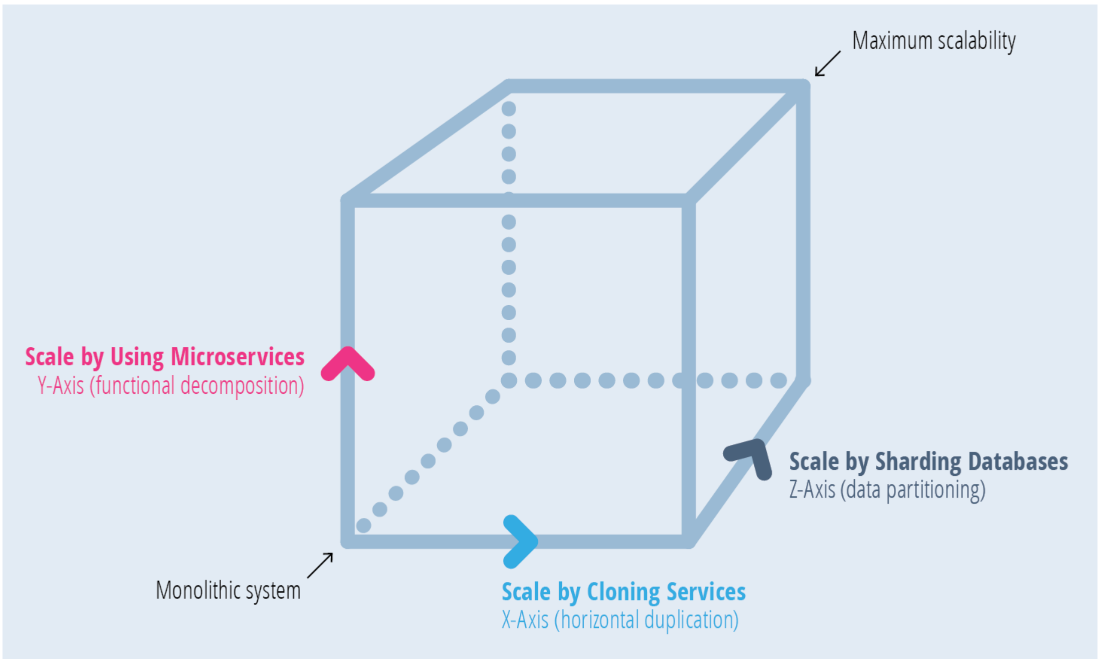
Maximum scalability
Scale by Using Microservices
Y-Axis (functional decomposition)
Scale by Sharding Databases
Z-Axis (data partitioning)
Monolithic system
Scale by Cloning Services
X-Axis (horizontal duplication)

Figure 6. Scale cube (adapted from Juneja 2016)

In general, a monolithic application can usually scale only in one direction (X-Axis). However, microservices have the edge over the monolith and can be scaled in all three directions, which gives them the leading position.

3.2.4 Software delivery

In microservice architecture, each microservice is maintained by a separate development team. The team consists of a specialist from each area (UI, middleware, backend, DBA) to cover a full stack of technologies for a successful software delivery. Each team has full control over its microservice and is also responsible for its deployment. Thanks to the small code base of microservice is development team able to act in very agile way and be able to respond to business needs very quickly. This enables to deliver new feature much more quickly, in some cases on a daily basis. Such ability is often needed in today’s market where there is high competition.

Frequent updates and new version delivery are areas where the classical monolithic architecture struggles. It is not so easy to fix or change parts of software with a huge code base. After each change, a sequence of regression tests needs to be run to ensure that the code changes do not modify the behavior of different parts of the software. This is especially true in cases when backward compatibility is needed and the monolithic application needs to contain multiple versions of the same module. In the world of microservices this issue can be avoided in an elegant way, where different versions of same microservice run autonomously side by side. (Goldsmith 2015)

In order to speed up the software delivery process, microservices must heavily rely on automation. This includes automated testing, continuous integration and continuous delivery. Having an automated build pipeline also brings greater reliability to deployment where manual deployment could fail just because of the human factor.

The theory behind software delivery is nowadays quite mature and popular. This set of practices is also called DevOps. Basic set of tasks in delivery pipeline can be seen in Figure 7.

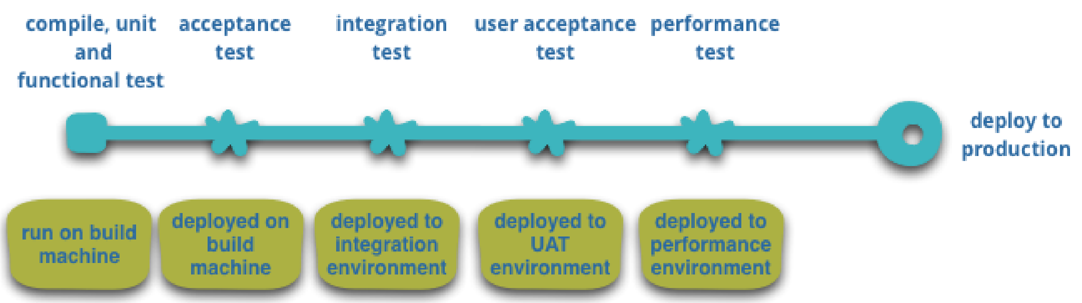
compile, unit and functional test
acceptance test
integration test
user acceptance test
performance test
deploy to production
run on build machine
deployed on build machine
deployed to integration environment
deployed to UAT environment
deployed to performance environment

Figure 7. Basic build pipeline (adapted from Lewis, & Fowler 2014)

3.2.5 Loose coupling

Venkat Subramaniam, as many others, mentioned in his talk at Devoxx Conference 2015 that one of the core software design principles is loose coupling (Subramaniam 2015).

Loose coupling in software design means that modules, services, classes and interfaces should have knowledge only about their own implementation details. This promotes using encapsulation and restricts access to software components as much as possible by defining clear and meaningful interfaces between them. (Subramaniam 2015)

By breaking the rule of loose coupling, software developers can end up with code that has great number of direct dependencies. Tight coupling makes code refactoring quite tricky, where the change made in one module can affect another module and break the whole application. This lowers productivity and developers' courage to make any changes in the software (Ranney 2016).

Microservices are by their nature more evidently separate ones from each other than modules in monolithic architecture. Microservice architecture promotes inter-service communication only via service APIs. It is also strongly recommended to use a separate database for each microservice. This not only allows each microservice to use the best suitable database technology for the given context and yet prevent the existence of dependencies via a shared database which could be less evident. (Ranney 2016)

The connection between database and microservice can be seen in Figure 8. It also shows how microservices can depend on each other through shared database, even though they do not seem to be related at the first sight.

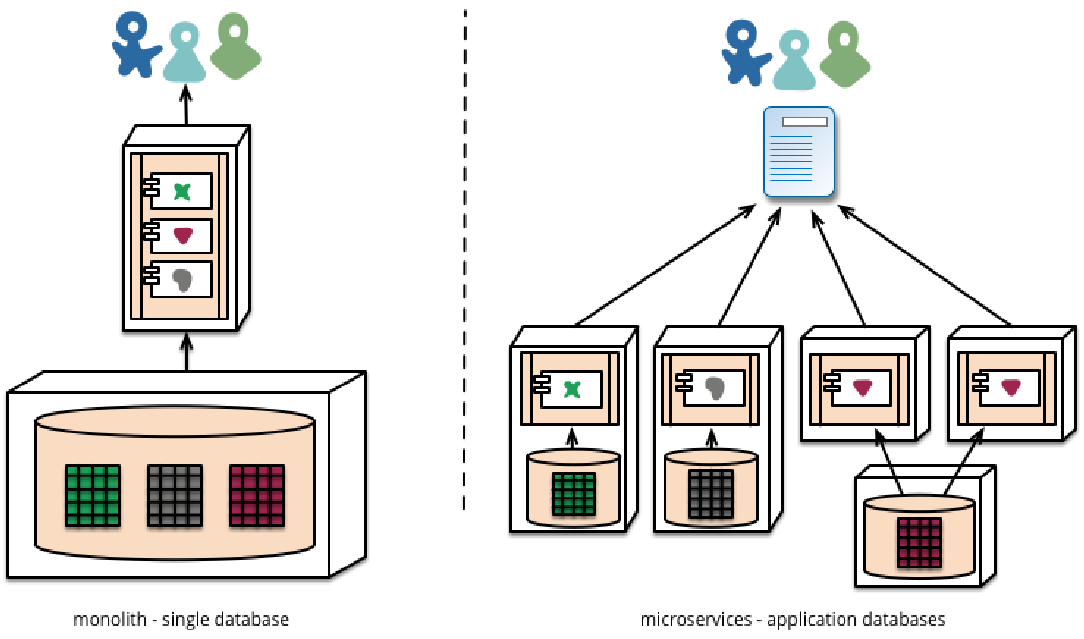
monolith - single database
microservices - application databases

Figure 8. Each microservice has its own database (adapted from Lewis, & Fowler 2014)

Keeping an implementation hidden behind API allows development teams to change code faster and deliver product updates very quickly (Goldsmith 2015).

To gain loose coupling, software architects designing microservices need to think with domain requirements in their minds. Bad separation of service responsibilities can result in two microservices tightly dependent on one another. A good rule of thumb is to keep related business responsibilities together within one microservice. This ensures high cohesion which is another core software design principle. (De Santis, Florenz, Nguyen, & Rosa 2016, 17)

It is not always easy to get the boundaries of microservice responsibilities right at the first time. Especially helpful can be the design principles of domain-driven design with its great emphasis on Bounded Context that delimits context and bounds within which the designed model applies (Evans 2015). Martin Fowler (2015) also emphasizes that having a good separation of responsibilities in microservice architecture is very important. He states that refactoring of functionality between services is much harder than it is in a monolith.

3.2.6 Performance

Benefits of running separate services in different processes do not come without any cost. The whole machinery behind transport protocols that deliver information from one service to another adds to the overall latency of the system.

Because of the higher latency caused by transport protocols, responsiveness of each microservice should be carefully measured. If the response time is not fast enough, performance optimization should be definitely considered. This is especially true for big applications where one microservice usually triggers several others.

As an example, microservice with average response time 1ms has at 1% of times slowdown to 1000ms. This microservice prepares data for a higher level microservice, which sends it to the user for rendering. If higher level microservice calls the slave microservice only once, the overall response time is very quick with 1% chance in slowdown. However, if it calls the slave 100 times, the chances that the user experiences slowdown jump to 63.4%. The rapid increase of slowdown chances can be seen in Figure 9. (Ranney 2016)

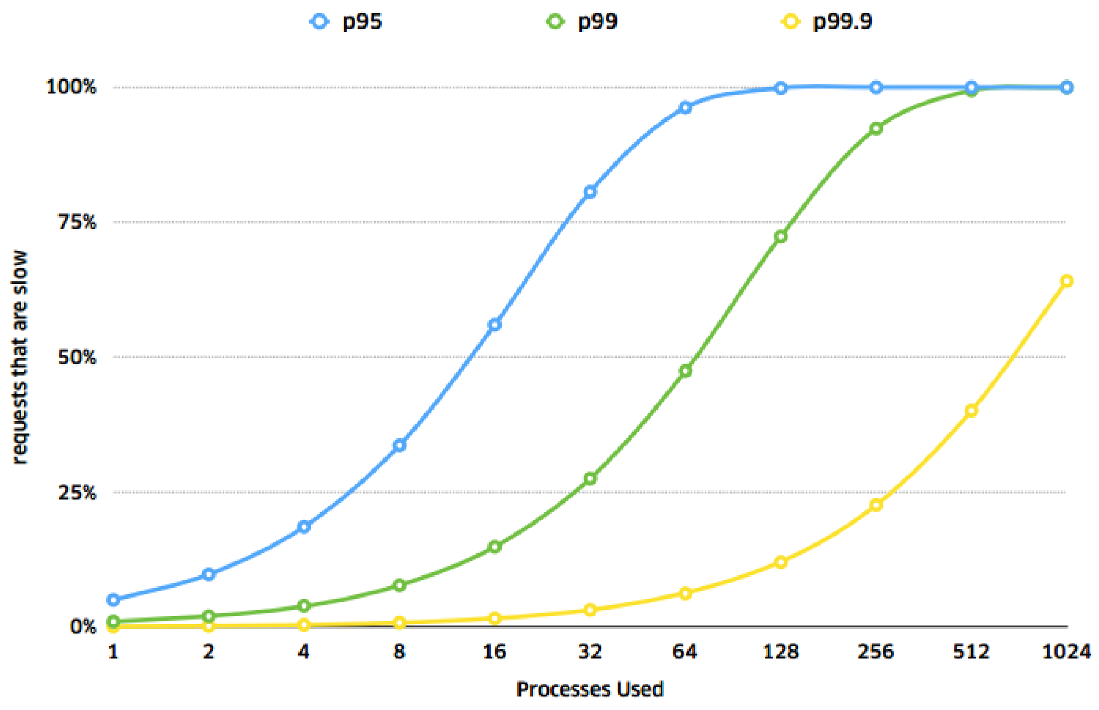

Figure 9. Slowdown chances for overall microservice response (adapted from Ranney 2016)

3.2.7 Communication

One of the areas where problems can occur in big microservice applications is communication. When one team needs to use services of other team's microservice, they use a defined API. Most widely used RPC in microservice world are HTTP or REST. Retrieving data via these protocols can sometimes be tricky. The responses usually come in formats as JSON or XML. JSON and XML are not typed formats which can make them error prone. Errors can be also made by omitting required custom request headers or through the fact that some teams may use different HTTP methods for same kind of action (e.g. HTTP PUT method should be used for data; however, some developers may use HTTP POST instead). (Ranney 2016)

In the world of Monolith, these problems basically do not exist because data are being passed via object messages. Type checking can be ensured by using statically typed programming language as Java.

3.2.8 Testability and monitoring

Microservice, thanks to its small size is very easy to unit test. However, unit tests are not able to test resilience of the whole system against sudden microservice failure. A tool called Chaos Monkey can be used for those test cases.

Chaos Monkey is a project created and open-sourced by Netflix. Its job is to randomly terminate instances of services in production environment to ensure that engineers have implemented services resilient to failure. (Chaos Monkey 2016)

From the monitoring point of view, microservices have a disadvantage over monolith. In a monolithic application, logging is usually done to the same file, therefore, all log messages can be found in one place and in chronological order. For microservices it is not possible to append log messages to the same file because they can be spread around multiple physical machines. In order to see a greater picture of the whole application, a separate system needs to be introduced for the centralized management of log messages. ELK Stack or Splunk is suitable for this job. (Ranney 2016)

3.3 Monolith first

Martin Fowler in his article *Monolith First* (2015) writes about an interesting finding. He says that most of the projects that he encountered as successful use, microservices are originally monoliths transformed over the time, which raises the question if creating a monolith with plan to transform it to microservices is not better that start implementing microservices at the first place.

This idea founds its arguments on fact that microservice architecture is more complex than the monolith one; thus, a monolithic application can be shipped more quickly. Another argument is that the key factor of good microservice design is to define the right boundaries between services. This requires deep knowledge of the business domain and is not an easy task to do even for an experienced architect. Bad design of microservices leads to greater problems than bad design on monolith. (Fowler 2015)

Implementing monolith first helps the development team to get to know the business domain. This knowledge can be leveraged to transformation towards microservices. The shift to microservices can be done two ways. The first option is to keep a solid monolithic base and start building microservices around it. The second option is to iteratively transform whole application to microservices. (Fowler 2015)

If one does not want to run into problems caused by bad microservices design or does not like idea of *Monolith First* strategy, it is possible to design coarse-grained microservices, larger than those one expects to end up with. This reduces risks of bad design and once the boundaries get stabilized, coarse-grained microservices can be broken down into finer-grained microservices. (Fowler 2015)

4 How to decompose Monolith to Microservices

4.1 Signs calling for change

When Monolith starts getting bigger and less and less manageable, common sense leads a development team to modernize the system in order to keep it live and competitive. Common signs that indicate a need for change are (De Santis et al. 2016, 36):

- Long deployment cycles
- No separation of modules (small change in code involves other modules to build, test and deploy)
- Legacy technology stack
- Scalability issues (inability to scale portion of application)
- It is hard to start on the project for new developers

The old system can be modernized in several ways. One of them could be to rewrite the whole code completely. This approach is radical and brings a great risk. Problems arise when old parts of the system need to be kept, which often means adding old bugs to the rewritten system. An alternative approach is to incrementally create a new system around the old one, until one day the old system will be replaced completely. (Fowler 2004)

4.2 Strangler Application

The name Strangler Application comes from rainforest vine. It climbs around the tree killing the host tree. This is an apt analogy to gradual adaptation of microservices besides monolithic core, which can result in a future shift to microservice architecture only. (Fowler 2004)

The main idea of this strategy is to delegate new features and functionality to microservices, while making most of the old monolithic application. This prevents monolith from growing bigger and less maintainable. It can be gradually broken up into new microservices until one day it disappears completely. (Fowler 2004)

New microservices created around an old core can be scaled independently and delivered separately to old modules. By adapting this approach, the development team makes the new system easier to modify in the future, where a once strangled system is easier to restrangle, which is not the case in replacing one monolith by another, newer one. (Fowler 2014)

The next three chapters discuss the steps that need to be taken while adapting Strangler Application pattern.

4.3 Request router

Request router is a key element of Strangler Application which helps to represent the monolith and new microservices around it as a single uniform application to the outside world. Its job is to send requests to monolith or microservice depending on the location of the implementation serving particular request. (Richardson 2016) Data flow managed by request router can be seen in Figure 10.

Monolith and new microservice only occasionally live next to each other without a need of accessing their data. To ensure that they can talk one to each other, glue code is needed and it can be provided in three ways (Richardson 2016):

- Allowing direct database access
- Maintaining its own copy of the data
- Creating a remote API

Allowing direct data access makes parts of the system dependent on each other. Allowing each, monolith and new microservice, to maintain its own copy of the data is possible; however, great effort is needed to ensure that the data are consistent across whole application. (Richardson 2016)

From these three options it is preferable to use the last one. Creating a remote API prepares the monolith to future decomposition to microservices where this change will be transparent to the communicating parts of the system. (Richardson 2016)

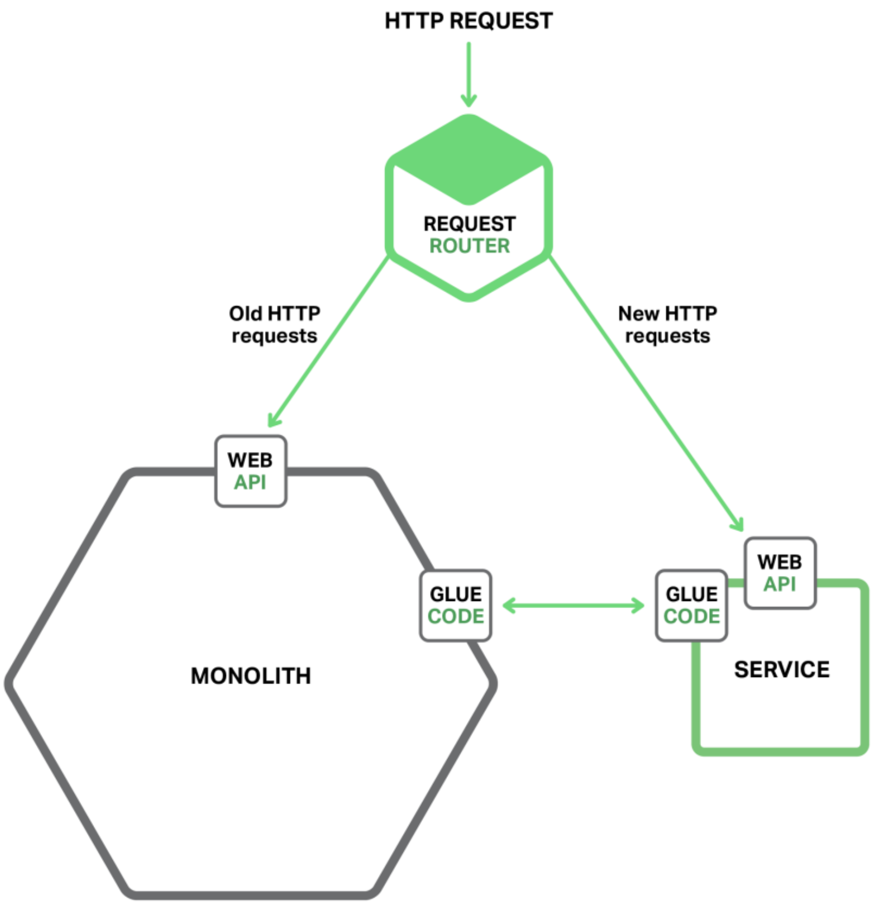

Figure 10. Request router (adapted from Richardson 2016)

4.4 Separation of presentation layer

Each monolithic application usually uses layered architecture. Each layer can access only services of the layers lying beneath it and should have no knowledge about layers sitting on top of it. This helps keeping logical parts separate and thus easier to modify. (De Santis et al. 2016, 28)

The message from good layered architecture is captured in Figure 11. It shows that if the integrity of each layer is not violated meaning only the upper layer has knowledge about the layers beneath it and no knowledge about upper ones, and direct communication is restricted only to the neighboring layers, a change of one layer should not cause changes in other layers.

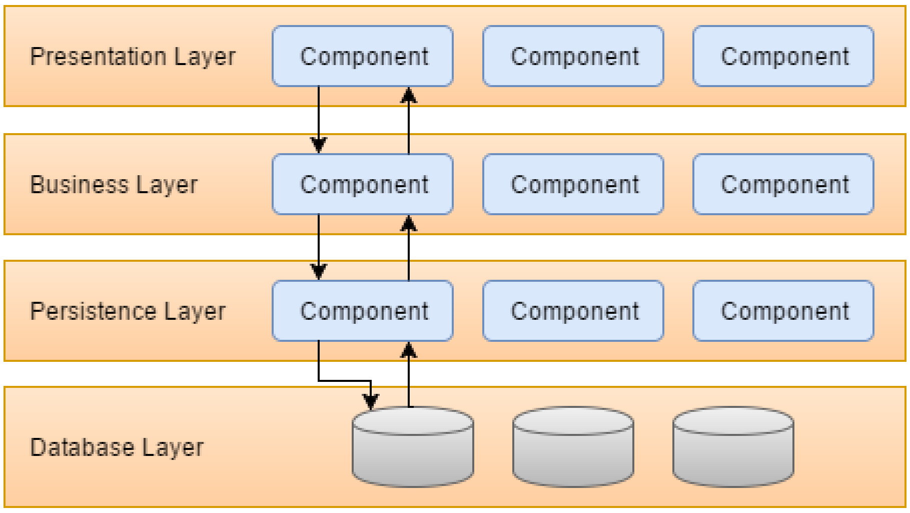

Figure 11. Layered architecture

Presentation and business layer are usually divided in a clear way. The separation of presentation layer to standalone application brings two major benefits. It allows the frontend of an application to develop independently to a monolithic core, which should speed up the deployment cycles and simplify the whole deployment procedure. Another benefit is that it forces developers to encapsulate the logic of the monolith to API that can be leveraged by new microservices developed in the future. (Richardson 2016)

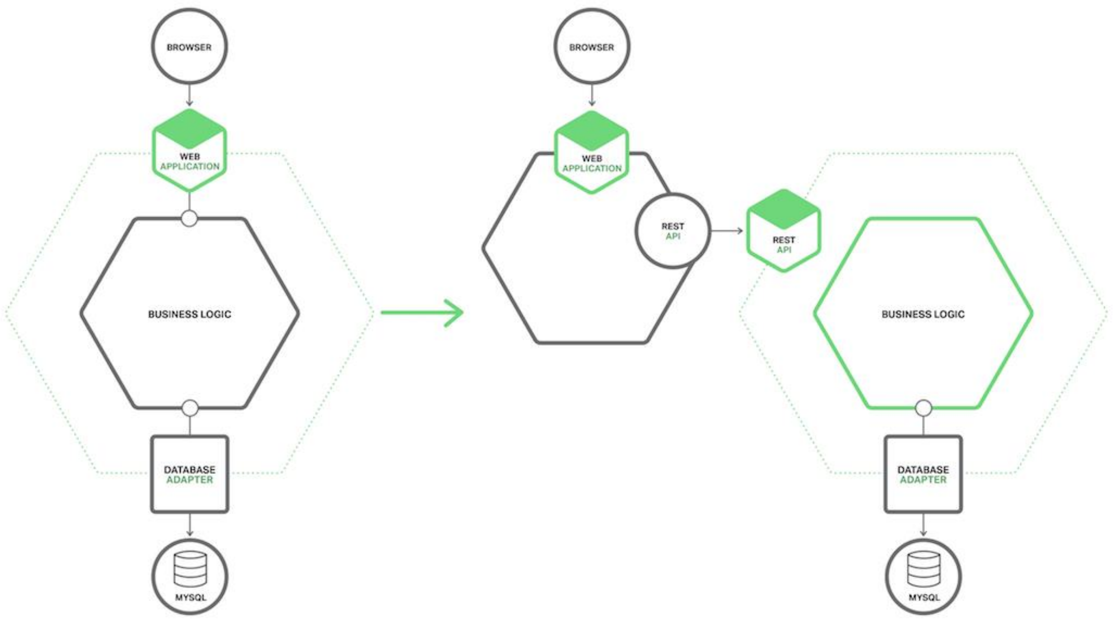

Figure 12. Separation of presentation layer (adapted from Richardson 2016)

Separation of presentation layer from business layer is displayed in Figure 12. On the left side are presentation and business layer together residing in one application. On the right side are those layers separate, communication via REST API.

4.5 Extract microservices

The final step of breaking up a monolith to microservices consists of two steps. In the first step, all dependencies between the monolith and the new service are untangled by defining the API between them. Once the API is defined, communication with the monolith can be restricted to API only, which prepares the service to be completely detached. (Richardson 2016)

In the second step, the service is moved to standalone microservice running in a separate process. After this step, a new microservice can be developed, scaled and deployed independently to monolith. (Richardson 2016)

The steps of separation module Z into standalone microservice can be seen in Figure 13. In the first step are introduced interfaces; through is which module Z is communicating with modules X and Y. Definition of the interfaces is the necessary step for making microservice standalone. In the second step are interfaces transformed to REST API and microservice is moved to separate process.

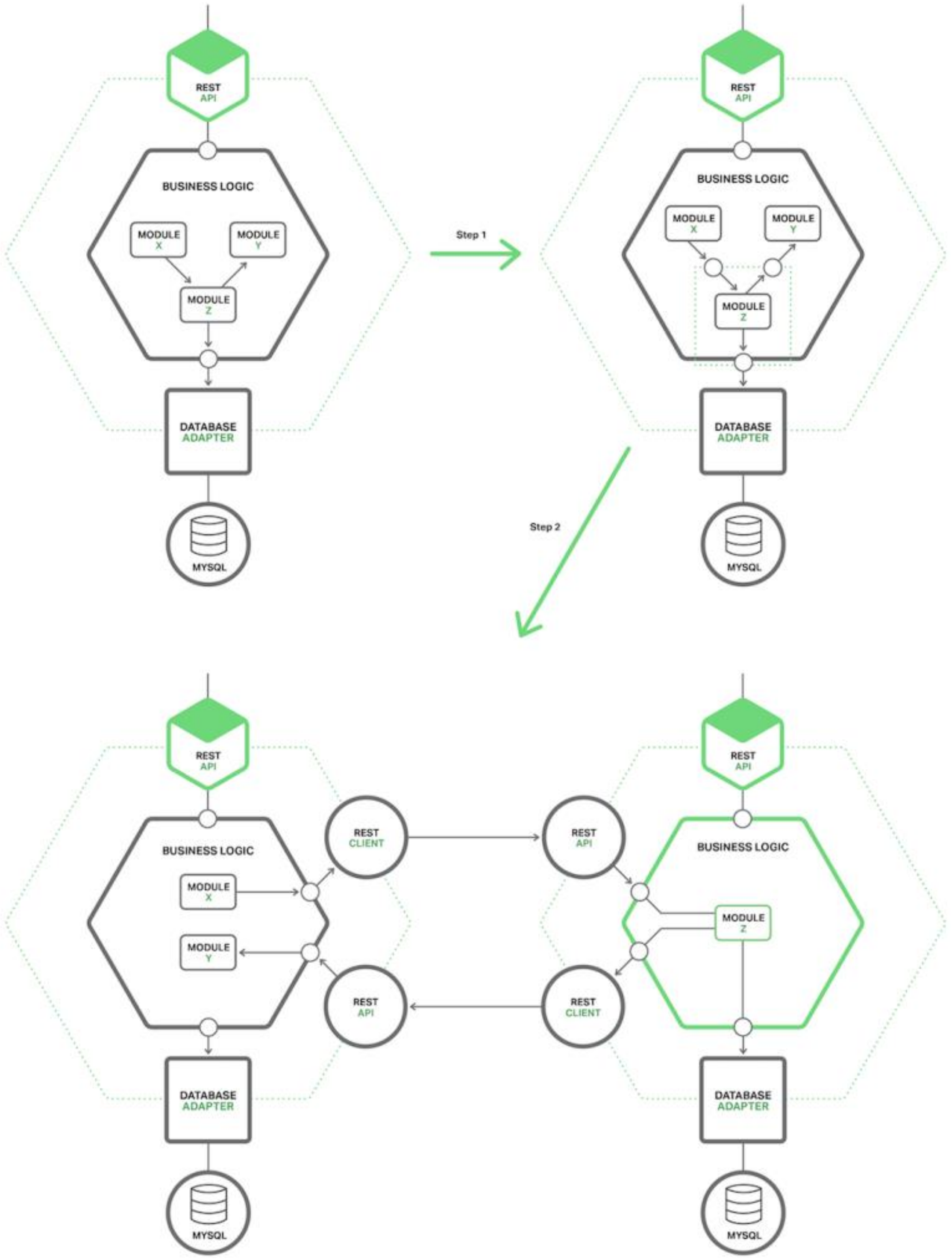

Figure 13. Process of breaking up monolith to microservices (adapted from Richardson 2016)

5 Implementation

5.1 Solteq Wellbeing project

5.1.1 Project objectives

Solteq, similarly to all modern companies, tries to care about their employees' wellbeing and offers various work benefits. Among those is also the possibility to get a massage and adjustment of working setup to keep correct ergonomy and prevent health problems caused by bad posture.

In order to make these services visible and easily accessible to all employees, the registration process needs to be available everywhere. With this idea in mind a small web application was designed and developed, where every employee can make a reservation for each of the available services.

The system also provides a nice user interface for the masseur. She can keep notes for each session and revise them later to improve the overall service and keep track of all previous procedures that the employee undertakes before.

All system requirements can be seen on use case diagram in Figure 14 below.

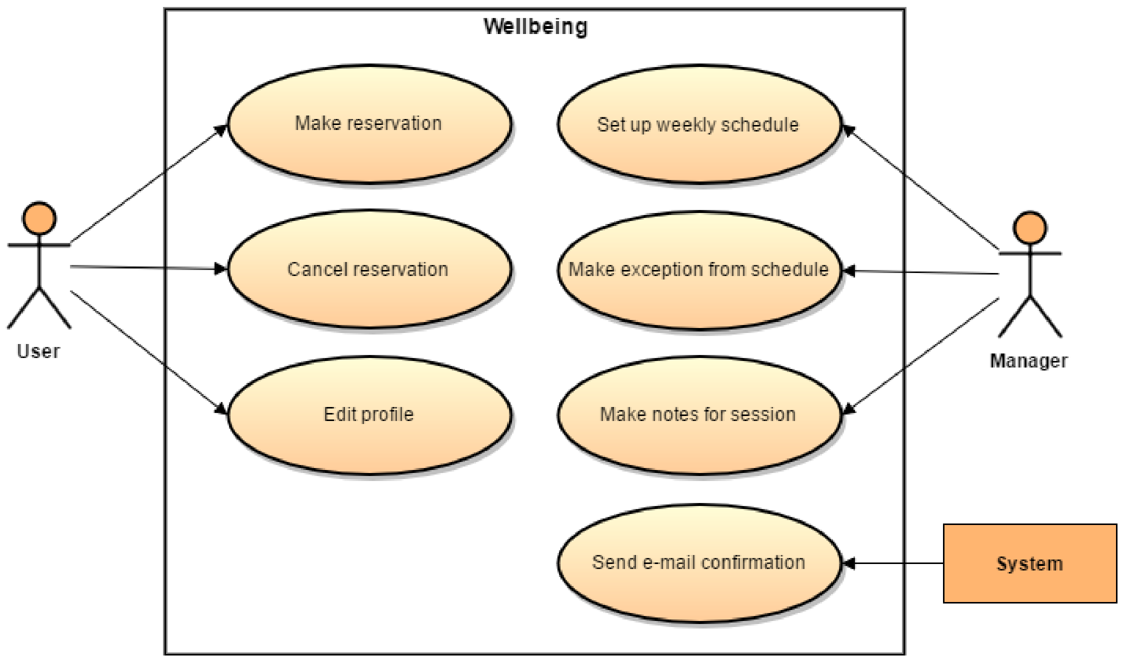

Figure 14. Solteq Wellbeing project: Use case diagram

Figure 15 displays the main user interface of Solteq Wellbeing application. There are tabs, one for each available service. The Calendar contains fields with appointment times available for reservation.

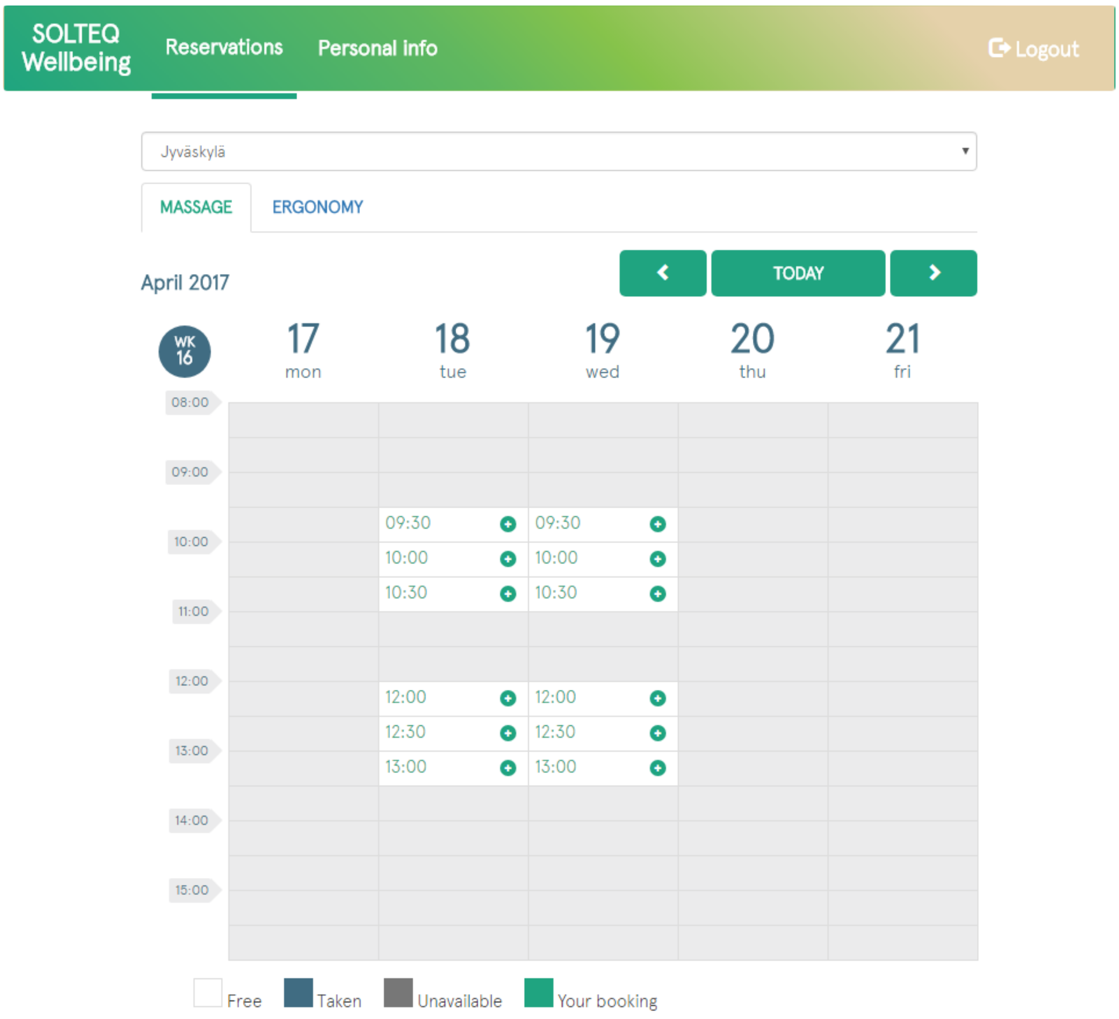

Figure 15. Solteq Wellbeing project: User interface

5.1.2 Technology stack

The whole application is developed using modern technologies. The frontend of the application is written in Angular JavaScript framework and works on all internet-enabled devices thanks to the responsive design.

The backend of an application is written Node.js using Express web framework. The application data is backed up by AWS DynamoDB, a document oriented NoSQL database.

The application is hosted on Amazon Cloud. Except the already mentioned DynamoDB, the application uses services of Elastic Load Balancing. Elastic Load Balancing keeps track of application health status and starts a new instance of application in a different availability zone, in case of failure. The very application runs on Elastic Compute Cloud (EC2 in the Figure 16.). The complete stack of used services is visualized in Figure 16.

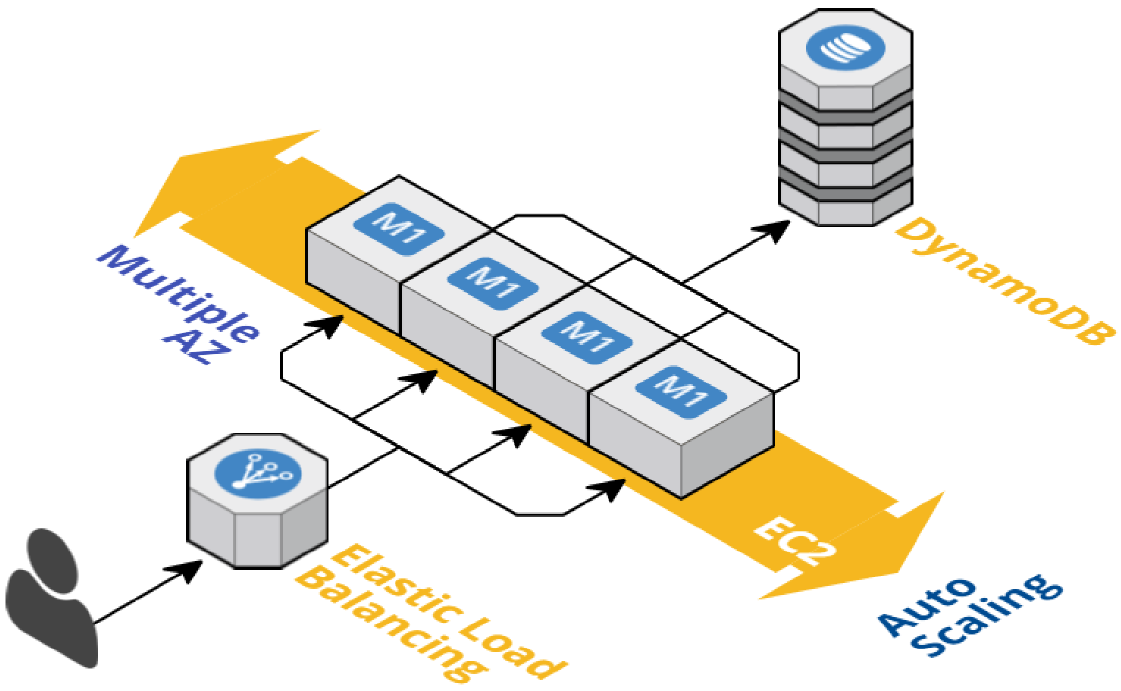

Figure 16. Solteq Wellbeing project: Used AWS services

5.2 Used AWS services

Amazon Web Services was chosen as a platform for implementation. This decision was made because it is currently the most popular cloud platform on the market and running application is already using some AWS services. Before diving into the implementation details, used AWS services will be described in this chapter, to make sure that reader is familiarized with those technologies.

5.2.1 Amazon S3 Bucket

Storage is one of the necessities in order to run application successfully. Applications need storage to save their state, user files and backups. Responsible storage solutions should provide data durability, high availability and easy upgradeability. To fulfill all these aspect is a simple task. Coming up with such solution on your own can be challenging task, resulting in high additional costs if not calculated correctly. Amazon S3 (Simple Storage Service) is solution specifically designed for these needs. (Amazon Web Services 2014)

Amazon S3 is object based (file based) storage designed for durability, high scalability and security at the low cost. Data in the S3 bucket are stored in multiple copies across multiple facilities to prevent possible data lost. Size of the one bucket is not restricted and can grow unlimitedly. (Amazon Web Services 2017)

Files can be automatically versioned by storing full copies at the specified time. Versioning can be used with combination of life cycle policies, that can automate data transfer to lower cost storage option and/or removed completely after specified amount of time. (Amazon Web Services 2017) Such life cycle policy can be seen in Figure 17 where uploaded files are automatically moved to infrequent access storage after 30 days. Then after another 30 days they move to archival storage and finally completely removed after 90 days from the time they were uploaded.

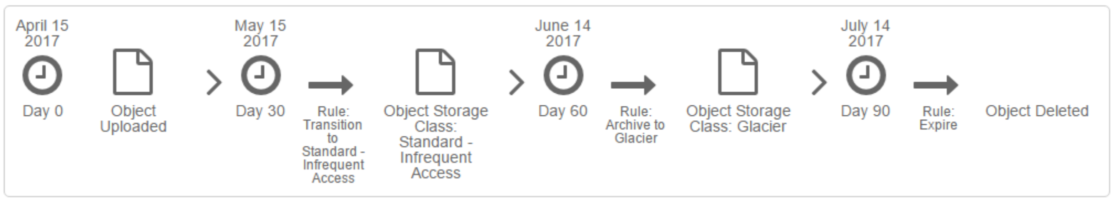

Figure 17. Amazon S3 Bucket: Life cycle policy (adapted from AWS Console 2017)

By default all the data stored on Amazon S3 are private. They can be exposed to public by specifying user or resource-based policies. Complete access history can be then reviewed in access log. (Amazon Web Services 2017)

Data in the S3 bucket can be secured by enabling automatic data encryption and using SSL to transfer filed from and to the bucket. (Amazon Web Services 2017)

Among other benefits, Amazon S3 bucket can be used for static web page hosting. This comes with high scalability and can be achieved for much lower price than equally scalable dynamic web page hosting option. (Amazon Web Services 2014)

Amazon offers S3 comes in the four different storage classes, listed by price from highest to lowest (Amazon Web Services 2017):

- Amazon S3 Standard
- Amazon S3 Standard – Infrequent Access
- Reduced Redundancy Storage
- Amazon Glacier

Standard solution is suitable when high availability is needed. If data are not accessed on regular basis, more suitable can be Infrequent Access option, which comes with lower availability and is charged for data retrieval. Easy reproducible data can be stored on Reduced Redundancy Storage, which offers lower durability for lower price. The last and the cheapest storage option is Amazon Glacier, which has high durability and is primarily meant for large data archive although, data stored on Amazon Glacier are not immediately accessible and can be available after 3 – 5 hours from initial access request. (Amazon Web Services 2017)

5.2.2 Amazon Route 53

Amazon Route 53 is a DNS service with high availability and scalability. Its main purpose is to setup routing rules between other AWS services such as Amazon EC2, Elastic Load Balancer, and Amazon S3 or outside of AWS. (Amazon Web Services 2017)

Amazon Route 53 is not only DNS but also a domain name registrar, and one can buy a domain name from within Amazon Web Services console. (Amazon Web Services 2017)

Routing rules can be configured using graphical user interface called Traffic Flow. Traffic can be managed based on (Amazon Web Services 2017):

- Response latency
- Geo location
- Weighted round robin

By choosing response latency based routing; traffic is evenly divided between multiple instances of web server based on server response time. Geo location based routing uses multiple web servers spread around the globe and serves web content to users from the nearest server based on the user IP address. Weighted round robin is based on “weights”, that are specified to each server and all user requests are then divided to servers based on the weight ratio. This comes very useful in A/B testing of an application where developers do not need to handle this routing on the application layer. (Amazon Web Services 2011)

Amazon Route 53 also enables users to configure health checks to monitor application endpoints and setup secondary content providers (e.g. static web page hosted on Amazon S3 bucket) to failover in case of application downtime. (Amazon Web Services 2017)

5.2.3 Amazon DynamoDB

Amazon DynamoBD is fully managed NoSQL database service with single-digit millisecond latency at any scale. It supports two store models: key-value store and JSON document store. Data are stored on SSD storage across three distinct geo locations. Amazon claims that DynamoDB is great fit for mobile and web, gaming, advertising and IoT applications. (Amazon DynamoDB 2017)

Amazon DynamoDB is schema-less what means, that one can add additional row of data to existing database table without any schema modification (Amazon DynamoDB 2017).

Database reads can be set up in two ways (Amazon DynamoDB: Read Consistency 2017):

- Eventually consistent reads
- Strongly consistent reads

Main difference between the two is that eventually consistent reads give the best performance, but does not provide immediate consistency. It means that one can receive old copy of data after immediate query of previously written data; however, data should be updated within one second after write. Strongly consistent reads ensures that query results reflect all previous writes that was successfully received; however this comes with performance penalty. (Amazon DynamoDB: Read Consistency 2017)

Amazon DynamoDB is charged for data size and throughput capacities. Capacity unit size is following (Amazon DynamoDB: Limits 2017):

- one read capacity unit is:
    - one strongly consistent read per second, or
    - two eventually consistent reads per second
- one write capacity unit is one write per second

Those capacity units need to be calculated and reserved for each table. If real world demands exceed reserved capacity units, performance of reads and writes is throttled. (Amazon DynamoDB: Limits 2017)

Main selling point of Amazon DynamoDB is one click scalability, where scaling up or down does not require any maintenance window and can be done simply by raising or lowering number of read and write capacity units reserved for database table. (Amazon DynamoDB 2017)

It also provides streams, which can be subscribed to and used as an event triggers on the database table. This topic will be covered in more detail in following chapter *AWS Lambda*.

5.2.4 Amazon EC2

In the chapter *Cloud computing* it was explained what benefits brought could to day to day life of the developer. There is no more need to provision your own hardware. Instead, one can leverage available compute power from the cloud vendors such as Amazon EC2.

Amazon EC2 (Elastic Compute Cloud) provides compute capacity in the form of a virtual machine. Amazon offers wide variety of hardware configuration to choose from, so that it can fit every application needs. There are currently multiple options in each of categories: storage optimized, memory optimized, CPU optimized and graphics optimized. (Kroonenburg 2017)

EC2 instance can be launched by choosing one of the AMIs (Amazon Machine Image). AMI is a template that contains software required to launch a new EC2 instance. It always contains operating system and in some cases additional software such as application server, database server, Docker and couple programming languages. Amazon currently offers 31 AMIs. One can choose from them based on operating system preference. Available operating systems are Microsoft Windows Server and Linux in many distributions (Ubuntu, Red Hat, SUSE and Amazon Linux). (Amazon Machine Images 2017)

Amazon EC2 used as a storage EBS (Elastic Block Storage) what is SSD or HDD based storage automatically replicated to gain higher durability and availability. Besides EBS, EC2 instances can use as storage EFS (Elastic File System). EFS is different from EBS, because it is automatically scaled to user needs up to petabytes, without need of terminating EC2 instance. It can also be mounted to multiple EC2 instances at once, what is not possible with EBS. Instance storage can be versioned by taking snapshots and secured by enabling encryption. (Kroonenburg 2017)

One of the nice benefits is that EC2's compute capacity is resizable; however it is not possible without any downtime. Hardware resources of each instance are monitored and can be used to configure health checks and alarms. Those can be used to automatically trigger new instances or execute different action e.g. send e-mail. Thanks to automatic resizing, applications can be scaled up and down based on current application load without intervention of the human factor. (Kroonenburg 2017)

Amazon offers EC2 in four different pricing categories (Amazon EC2: Pricing 2017):

- On-demand
- Spot instances
- Reserved instances
- Dedicated hosts

On-demand instances are the best fit for scenarios with unpredictable application load where is scaling needed based on the use of the application. In this model is customer charged for what he uses. Reserved instances are cheaper than on-demand instances, but are available only in long period contracts, thus are suitable only for applications with stable application load. Spot instances allows customer to bid the price he is willing to pay for compute power he wants. If customer's offer is accepted, he will get demanded computational power (usually during nighttime when compute power of servers is not utilized as much as during the day). This pricing model is suitable for applications with flexible start and end times or when running application is feasible only at very low cost. Dedicated hosts are used in scenarios when customer needs full dedicated physical machine for his application. (Amazon EC2: Pricing 2017)

5.2.5 AWS Lambda

AWS Lambda is a service that allows developers to run the code on a scale without need of provisioning and managing servers. All servers' infrastructure and running environment is handled by AWS. Thanks to this, one can run his application on a scale without any administration. (AWS Lambda 2017)

AWS Lambda offers a convenient way how to develop event-driven application. Amazon integrates it with other AWS services such as Amazon S3 bucket, Amazon DynamoDB, Amazon API Gateway and many others. These services provide events that can be intercepted by AWS Lambda which executes the code in a response to them. (Invoking Lambda Function 2017)

One of the use cases where AWS Lambda can be used is DynamoDB trigger. It can be seen in Figure 18 where is AWS Lambda subscribed to DynamoDB stream and can invoke an action as a response to table row insert, modification or removal.

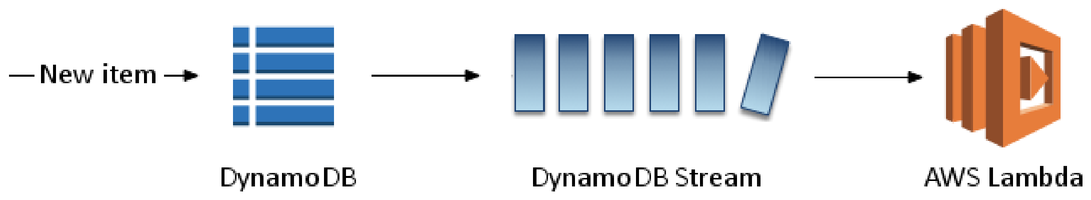

Figure 18. AWS Lambda responding to DynamoDB insert (adapted from Amazon DynamoDB: Processing New Items in a DynamoDB Table 2017)

Other common use cases of AWS Lambda is real-time data processing (e.g. creating image thumbnails on an image upload to Amazon S3 bucket or database replication across multiple geo locations) or as a backend of an application in combination with Amazon API Gateway. (AWS Lambda 2017)

AWS Lambda currently supports four different programming languages: JavaScript (Node.js), Java, C# and Python. After code upload, AWS Lambda is monitored by Amazon CloudWatch that keeps logs and metrics of each AWS Lambda execution. (AWS Lambda 2017)

All the new opportunities that AWS Lambda brings to the table come with appealing pricing where user is charged only for number of invocations and execution time of a function. This can result in great money savings especially for applications with the low user base. (AWS Lambda 2017)

5.2.6 Amazon API Gateway

Amazon API Gateway is a fully managed service that allows users to expose their business logic running on EC2 instances, AWS Lambda or any other application via RESTful API. (Amazon API Gateway 2017)

It supports running multiple versions of API at the same time, what can be leveraged in testing and pushing new releases. Whole API is monitored by Amazon CloudWatch. Data tracked by Amazon CloudWatch can be later on used in the process of examining performance and latency issues. (Amazon API Gateway 2017)

Amazon API Gateway also supports authorization an access management for granular access control management to individual API endpoints, which helps keeping authentication logic separate from business logic and helps in overall code understandability. In the process of API development are equally important mocking abilities that can rapidly help in the development efficiency. (Amazon API Gateway 2017)

Configuration of the API is straight forward and can be set up in minutes using Amazon Web Services console. If one does not prefer using AWS console, API Gateway also supports import and export using Swagger definition files, which is currently one of the most popular API frameworks out there. (Import and Export API Gateway API with Swagger Definition Files 2017)

Amazon API Gateway is meaningfully priced where users pay only for number or requests and amount of data transferred out from the API. (Amazon API Gateway 2017)

5.3 Microservices design

The previous chapter *Microservices* pointed out what benefits microservices have over monolithic architecture. To fulfill the basic requirements of microservices, the old application structure needs to be broken down to smaller individual parts that live independently one to each other. These parts should be then deployed independently and run in the separate processes. In the first step of the design it should be decided which technology will be used as a basic building block for microservices.

One possible solution would be to run multiple Amazon EC2 instances and have all microservices living on them. Traffic between microservices would be divided by using load balancing technologies. In the case of high system load, should be introduced new Amazon EC2 instance with additional microservices to divide overall system load more evenly. This approach does not bring many benefits over the existing monolithic solution, and it is very complex to manage from load balancing point of view.

A better solution is to leverage the existing cloud options and pick FaaS as a building block of the new microservice architecture. By choosing AWS Lambda as a running environment of one microservice, load balancing management burden is left on the shoulders of cloud providers. User can enjoy endless scaling without need to worry about underlying infrastructure.

Another benefit of AWS Lambda is its cost in comparison to traditional Amazon EC2 instance. Amazon EC2 instance is priced based on its hardware configuration, whereas AWS Lambda is charged by number of requests and execution time of individual functions. Current monolithic application is running on EC2 instance with 1 CPU core and 1.7 GB RAM what makes what makes $ 33.84 for a moth. AWS Lambda is drastically priced where 1M or requests costs $ 0.2 and 1 GB/s of compute power is $ 0.00001667. (AWS Lambda: Pricing 2017) If this price difference is reflected to a real world application such as Solteq Wellbeing project, it turns out that using Amazon EC2 as a platform for the application is much more expensive that AWS Lambda. As an example to demonstrate the price benefits of AWS Lambda over Amazon EC2 consider that each employee checks application on a daily basic and executes normal workflow, which means:

- Check calendar for massage 3 weeks ahead
- Check calendar for ergonomy 3 weeks ahead
- Make a reservation in the system

This workflow results in 20 API calls in total for one employee and 90000 calls for whole company in one month (considering that company has 150 employees and they use system also during weekends). This load AWS Lambda costs will result in $ 0.25 for a month (supposing that one AWS Lambda used 512 MB of RAM and has execution time 300ms). The price difference between the two can be seen in Figure 19 where the X-axis represents the number of API calls, and one AWS Lambda execution time is 300ms with RAM usage of 512 MB.

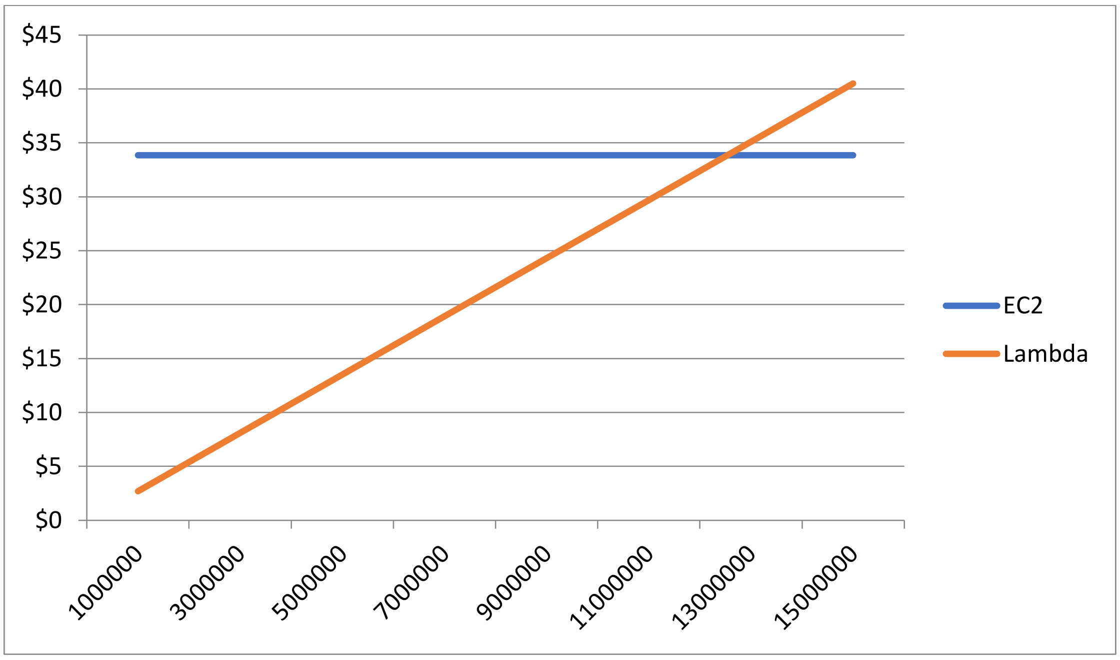

Figure 19. Price comparison of Amazon EC2 and AWS Lambda

One of the key concepts of microservices is that each microservice should contain only one element of the business domain acting in its context encapsulated within its boundaries (Fowler 2015). Access to this unit of a business domain is exposed to other microservices only via microservice API. In the design of decomposition to microservices is needed to gradually peel microservices that can be separated from the monolithic application. To do so correctly, it is need to get to know business domain of an application.

In Figure 20, RESTful API of the existing application can be seen. Each API call is labeled by its method type and colored red if it can be executed only by a manager user role (those API calls are not available for a standard user).

The main purpose of the application is to inform users about available reservation times for each one of the services and make it possible to reserve a session for that service in one of the free time slots. Services may vary across different company offices, which is why they are specified for each city (office) specifically.

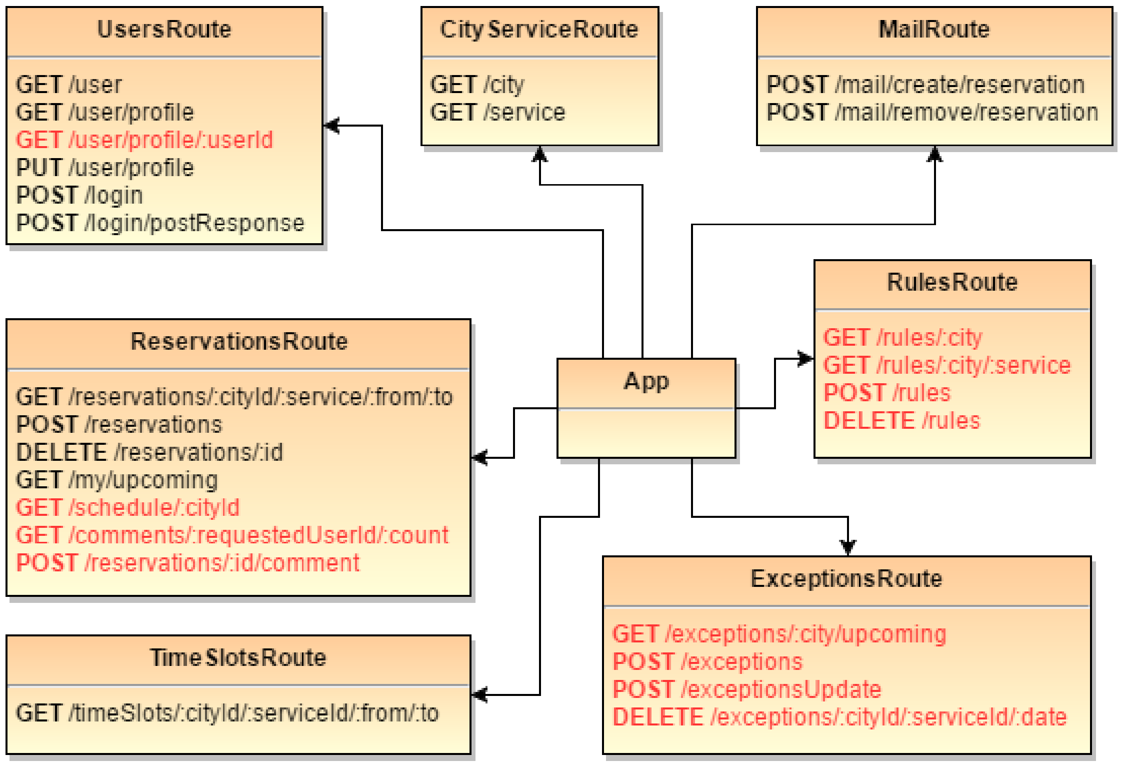

Figure 20. Solteq Wellbeing API

The system also supports a user interface for the masseur (manager). She can redefine the weekly schedule by modifying time rules that hold information, when in the work week are services planed. If a special occasion occurs or something causes that service cannot be performed in a predefined time slots, she can create an exception from the weekly schedule. An exception from the weekly schedule is tied to a specific day and applies only for that day.

When user asks the application to show the upcoming schedule, it is rendered by combining all three pieces of information: service time rules, exceptions from the weekly schedule and reservations made already. User can then choose one of the untaken time slots and make a reservation. If the reservation succeeded, the user receives an e-mail notification.

In Figure 20, all API calls are grouped together by their logical position in the application business domain. Each logical group embodies one route of the server.

When implementing the code for AWS Lambda, the natural structure of the application proposes to keep one Lambda function for one API call. This approach is also called “nanoservices”. Besides nanoservices, one can implement API endpoints in AWS Lambda as microservices or even monolith. Monolithic AWS Lambda basically means that the whole application logic is implemented and served from a single function. Microservices approach to AWS Lambda implementation stands right between nanoservices and monolith and keeps separate logical parts of an application together inside one Lambda function. (Nanoservices, Microservices, Monolith — Serverless architectures by example 2017) In transformation to microservices of Solteq, the Wellbeing application was chosen for this approach with a slight modification. Because of the application spare use, some logical parts of application that could easily stand in separate AWS Lambdas will be sharing one common Lambda function. The reason behind this decision is that AWS Lambda as a code does not reside in the physical memory of the server all the time, which means each first invocation of AWS Lambda experiences speed penalty caused by the fact that the code needs to be firstly transferred to the physical memory before the actual execution. This first function invocation is also denoted as cold start. After cold start, Lambda function stays in the physical memory, ready to be executed next time and is then again removed from it after a specific time threshold. (Smith 2017) By keeping the number of Lambda functions low, the user experience of the application will not feel so sluggish. The actual separation of API calls to standalone AWS Lambdas can be seen in Figure 21.

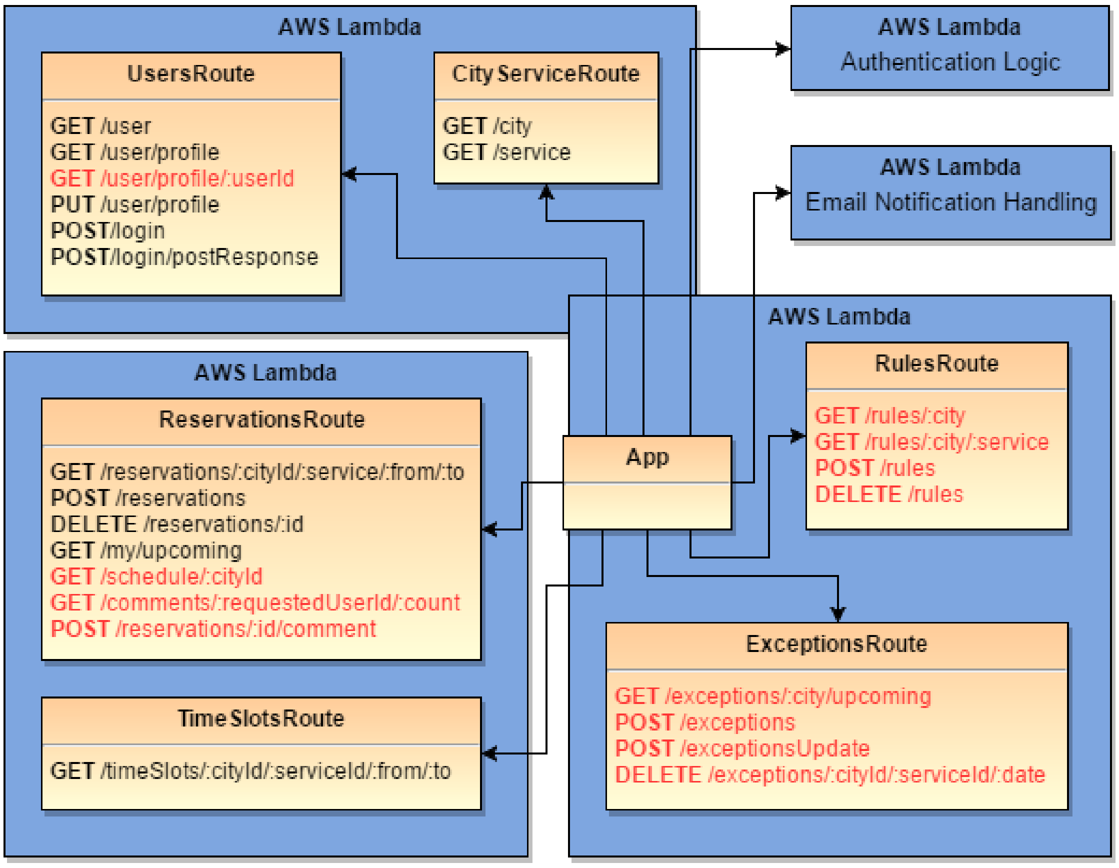

Figure 21. Solteq Wellbeing API moved to AWS Lambda

Instead of just moving the whole application logic to AWS Lambda functions, decomposition to microservices opens room for improvements. In order to expose application logic in Lambda functions, one needs to use Amazon API Gateway. Amazon API Gateway allows implementing one separate AWS Lambda as a handler of the user authentication. By doing so, the rest of the application can focus on executing the domain needs without worrying about user access rights. This can dramatically simplify the code and thus reduce the overall complexity.

As a second improvement, sending an e-mail notification can be simplified. By using Amazon DynamoDB streams, one AWS Lambda can subscribe to database table changes and response accordingly by sending an e-mail notification. This modification removes few API calls from original application API and makes it more cluttered. Both these changes are displayed in Figure 21.

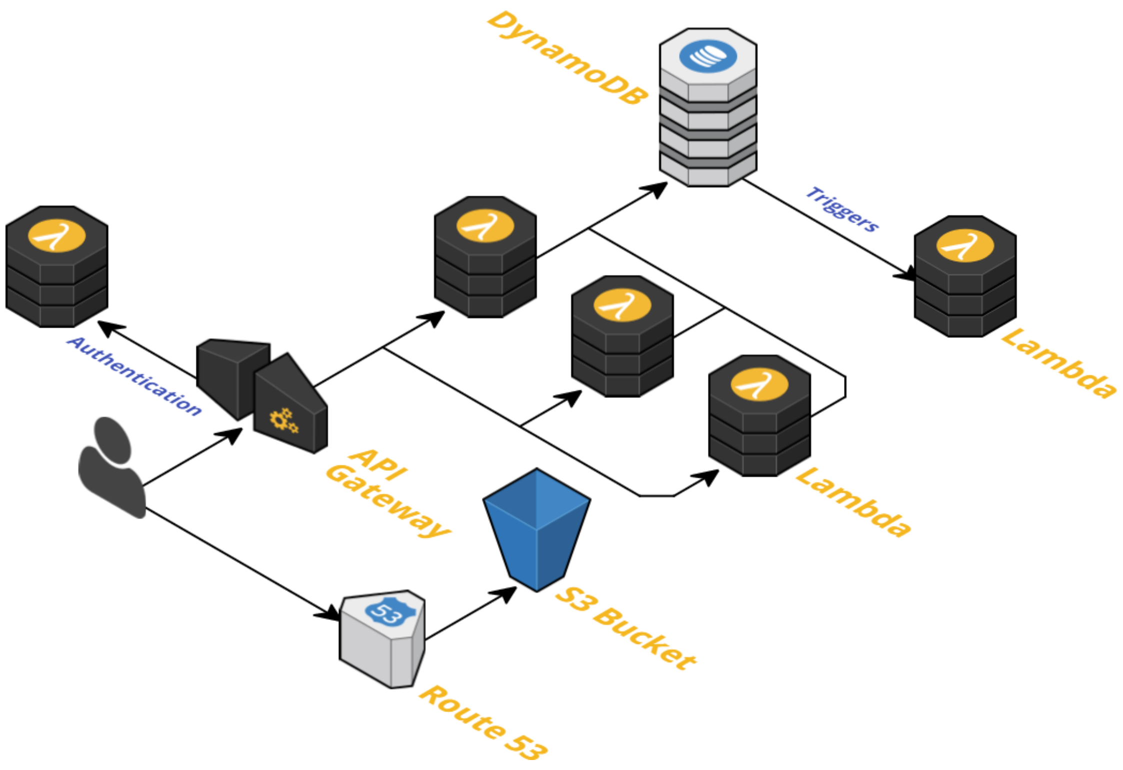

Figure 22. Solteq Wellbeing: AWS Lambda architectural design

If all these changes are translated to Amazon Web Services, it will look as seen in Figure 22: the application logic will be completely moved to AWS Lambda functions and composed to a single API using Amazon API Gateway. Amazon API Gateway will be secured by a separate AWS Lambda function that separates authentication logic from the rest of the application. Static web content does not need an application server, thus it can be stored on Amazon S3 bucket. User will receive it by accessing the application URL address through Amazon Route 53. As a data store will be used the same Amazon DynamoDB with running data stream that will be subscribed by AWS Lambda the purpose of which is to send e-mail notifications to users.

5.4 Transformation to microservices

5.4.1 Separation of presentation layer

The previous chapter showed and justified how the Wellbeing application could benefit from architectural transformation to microservices. The transformation of every application, even the small one, is usually a gradual task and requires multiple small steps until the expected result is met. Individual deployment of the smaller changes brings lower risk of failure, whereas the execution of one huge change can be quite hazardous. (Fowler 2004) This contrast was explained in chapter *Strangler Application* where a gradual change of an application was highlighted as a better option. For that reason, Strangler Application strategy was chosen for decomposition of Wellbeing application. Following chapters explain what technical steps were done to reach this goal.

All the changes needed to be done for the architectural change took place in the backend of the application. In the old design, frontend and backend of the system were living together, inside one Amazon EC2 instance. Because of this, it was wise to first split those two concerns before making any changes to the backend.

Thanks to the existing technology stack used on the project, the separation of the frontend from the backed application was quite straight forward. The whole frontend was written in Angular JavaScript framework and communicates with the backend through a defined API. The only measure that needed to be done to separate such presentation layer from the rest of the application was to spin a new web server to serve web content to the users.

Thanks to the fact that the frontend of an application is only static web content, it can be hosted on Amazon S3 bucket for a very reasonable price. In order to get the frontend of the application on the Amazon S3, a new bucket needed to be created, access permissions to the bucket contents set up, which document is the index of the web page specified and all the files to newly created resource uploaded. (Configure a Bucket for Website Hosting 2017) Even though this process may seem very easy to do, it is always good to automate as many things as possible in order to avoid mistakes caused just by the human factor.

To automate the creation of AWS resources, AWS CloudFormation can be used. AWS CloudFormation is a free of charge service allowing users to declaratively specify AWS resources in the single JSON configuration file. Configuration file can be uploaded to AWS, where then CloudFormation creates the all specified resources. (AWS CloudFormation 2017)

Figure 23 shows AWS CloudFormation file that creates Amazon S3 bucket and sets up all required permissions for web hosting.

```
{
  "Description": "Wellbeing: S3 Bucket for static web content",
  "Parameters": {
    "BucketName": {
      "Type": "String"
    }
  },
  "Resources": {
    "WellbeingS3Bucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "AccessControl": "PublicRead",
        "BucketName": {"Ref": "BucketName"},
        "WebsiteConfiguration": {
          "IndexDocument": "index.html"
        }
      }
    },
    "WellbeingS3BucketPolicy": {
      "Type": "AWS::S3::BucketPolicy",
      "Properties": {
        "Bucket": {"Ref": "WellbeingS3Bucket"},
        "PolicyDocument": {
          "Statement": [
            {
              "Effect": "Allow",
              "Action": "s3:GetObject",
              "Resource": {"Fn::Join": ["", ["arn:aws:s3:::",
                          {"Ref": "WellbeingS3Bucket"}, "/*"]]},
              "Principal": "*"
            }
          ]
        }
      }
    }
  }
}
```

Figure 23. AWS CloudFormation configuration file

This file consists of three parts: *Description, Parameters* and *Resources*. *Description* carries information about CloudFormation script and helps to identify it in the AWS console. *Parameters* specifies list of parameters expected to be filled in on CloudFormation execution. In this example, it has only one parameter, the name of the bucket, to be created. *Resources* part contains all the resources that need to be created. WellbeingS3Bucket resource represents Amazon S3 bucket publicly accessible for read. WellbeingS3BucketPolicy represents access rights for all contents of the WellbeingS3Bucket. It allows all bucket files to be publicly retrieved for read.

After creation of all these resources, upload of the web site contents is just a matter of one copy command (AWS CLI: Amazon S3 cp 2017).

5.4.2 Request router

After a successful separation of frontend and backend, the next step on the way to microservices was to define the request router. The main purpose of request router is to define a single uniform endpoint exposed to presentation layer. Without the presence of this necessary layer, the application could end up in a more complex situation, where a new module would be needed on the side of the presentation layer. The job of this module would be solely the decision making from which resource the presentation layer should ask data to render a response for the user request. Such an approach can be marked as an example of bad design where the rule of loose coupling is broken and a change in the backend side requires changes in frontend side of the application (Subramaniam 2015). Apart from this, the request router acts as a single point of access to data from the backend side of the application. Changes in backend implementation will not result in a change of the request router API, and thus are not reflected in the change of the frontend.

In the process of transformation to microservices serve this purpose Amazon API Gateway. It hides the differences between the old application implementation residing on Amazon EC2 instance and the new implementation running on AWS Lambda. User communicates solely through the interface of the API Gateway, and the background architecture behind it is not exposed to him. The communication flow described here can be seen in Figure 24.

Amazon API Gateway similarly to Amazon S3 can be created using Amazon CloudFormation. This functionality is already included in Serverless framework that is described more closely in the following chapter.

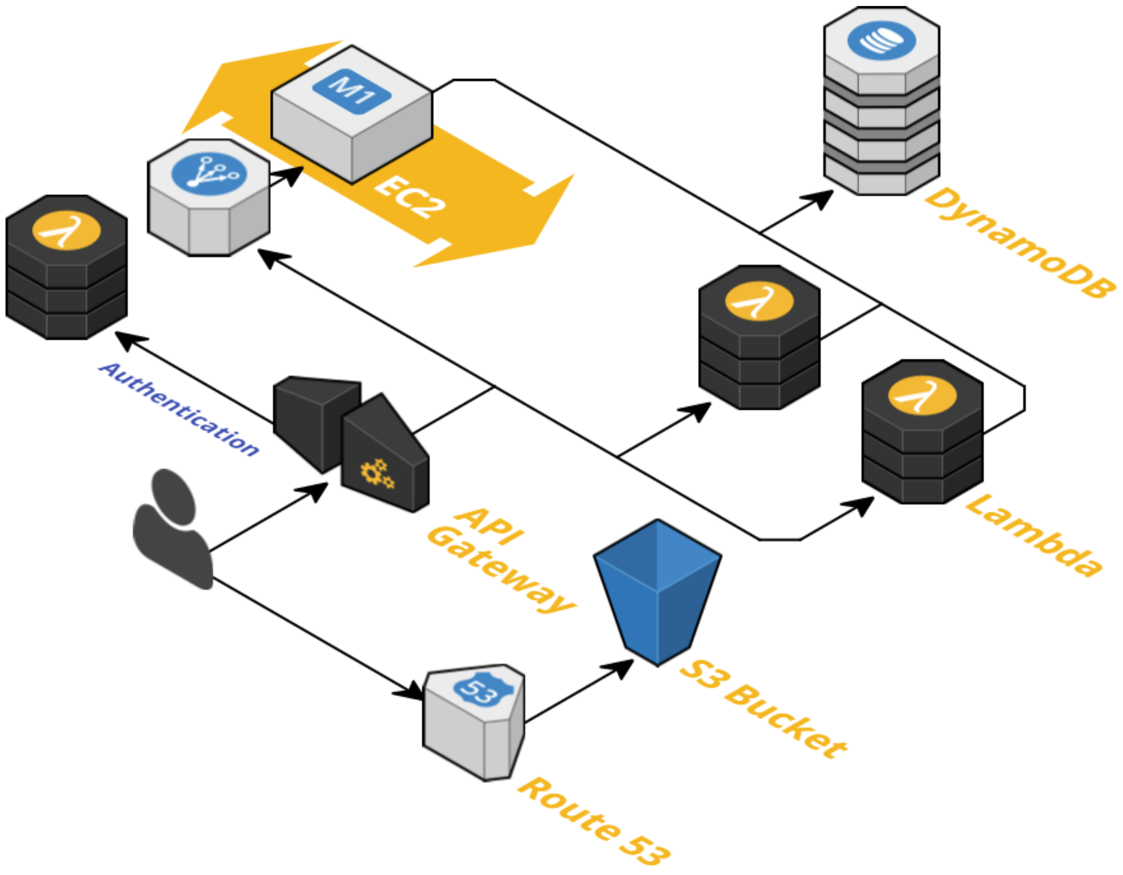

Figure 24. Request router: communication flow

5.4.3 Separation of modules

After the separation of the presentation layer and definition of the request router, the only part left is to create actual microservices. In the development process of microservices using AWS Lambda functions was leveraged simplicity of Serverless framewok.

Serverless is an open-source framework built to simplify the process of FaaS application deployment (Serverless framework 2017). It has a simple minimalistic command line interface and simplifies the work with AWS resources where AWS CLI commands are more verbose. The resource creation and configuration is done via single *serverless.yml* configuration file. It supports AWS CloudFormation resource definition syntax and uses AWS CloudFormation under the hood for resource creation. A sample configuration file can be seen in Figure 25. The file starts by specifying *service* name and continues by *provider* configuration. There can be configured implementation language of AWS Lambda functions, deployment region, stage of the project and other container specific configurations (Serverless.yml

Reference 2017). *Functions* section of the file specifies concrete AWS Lambda functions. Every function is named with a unique name (auth, user, mail) and must specify *handler* that is the relative path to the function file followed by the function name inside that file. Each function can respond to multiple *events* that starts the function execution. For example, the function *user* responds on Amazon API Gateway endpoint with HTTP POST method and /login path. HTTP events can be secured by specifying *authorizer,* another AWS Lambda that executes before HTTP event and decides if user is authorized to execute the requested method. Besides Amazon API Gateway, functions can be triggered by other events such as Amazon DynamoDB table change. At the bottom of the configuration file configured *mail* function can be seen that listens on Amazon DynamoDB table stream. The table must be specified by a unique ARN (Amazon Resource Name). One way how to securely specify the implementation and stage specific variables is to use environment variables. Each function has its own set of environment variables.

```yaml
service: wellbeing
provider:
  name: aws
  runtime: nodejs6.10
  stage: ${env:STAGE}
  region: ${env:AWS_REGION}
functions:
  auth:
    handler: functions/auth/index.handler
    environment:
        TOKEN_SECRET: ${env:TOKEN_SECRET}
  user:
    handler: functions/user/index.handler
    events:
     - http:
        path: login
        method: post
     - http:
        path: user
        method: get
        authorizer: auth
  mail:
    handler: functions/mail/index.handler
    environment:
        MAIL_HOST: ${env:MAIL_HOST}
        MAIL_PORT: ${env:MAIL_PORT}
        MAIL_USER: ${env:MAIL_USER}
        MAIL_PASS: ${env:MAIL_PASS}
    events:
     - stream: ${env:TABLE_RESERVATION_ARN}
```

Figure 25. Serverless framework configuration file

The individual AWS Lambda function should be small and simple enough to be easy to unit test. A good example for such function is the *auth* function (Figure 26) mentioned in the Serverless configuration file above. Its job is to validate the authorization token from the HTTP request head. If the authorization token is valid, the function returns the policy document that allows user to receive the response of the requested HTTP endpoint. Otherwise it returns an error message, saying that user access was not granted. Thanks to this authorization function, the whole logic hidden behind user access rights is moved outside of the application business logic, which brings more clarity to the overall code that has no knowledge about user authorization.

```
'use strict';

const jwt = require('../lib/JWT');
const { dynamodb } = require('../lib/DynamoDB');
const { getPolicy } = require('../lib/PolicyComposer')

module.exports.handler = (event, context, callback) => {
    const token = event.authorizationToken;
    jwt.verify(token, (err, payload) => {
        if (err) {
            getPolicy(null, event.methodArn)
                .then((policy) => callback(null, policy));
        } else {
            getPolicy(payload.email, event.methodArn)
                .then((policy) => callback(null, policy));
        }
    });
};
```

Figure 26. AWS Lambda: Authorization function

Besides the changes in authorization, another simplification was executed by using Amazon DynamoDB streams, where the sending of the e-mail notification does not need to be initiated from application logic as before but can respond to the events created by DynamoDB table insertions and deletions. The theory on how DynamoDB streams work was described in the previous chapter *AWS Lambda.*

The rest of the application business logic was unchanged and only requires to encapsulate already the existing code to new AWS Lambda functions.

6 Conclusion

The main objective of the bachelor's thesis was to examine what the differences between microservice and monolithic application are. The second step after gaining a better understanding of the benefits and obstacles brought by microservices to the design of an application, the transformation of the already existing monolithic application to microservices was executed. The cloud platform chosen for transformation was Amazon Web Services.

During the time of writing this thesis, I learned many new things. My knowledge of web development was very limited and mostly focused on the old way of doing things in the past. The topic of microservices got me interested and helped me to dive into the learning of new technologies. On my way I learned to use technologies such as Node.js, Docker and Amazon Web Services. It was particularly the Cloud technologies provided by Amazon that served me as an eye opener for all the possibilities nowadays available for developers.

The actual design of transformation to microservices was rather quite a straight forward task. I did not come across any major issues, most likely due to the small size of the application and the very modern technology stack used in development. Even though the design part of the transformation process was clear, I come across some small obstacles during the execution of the transformation. Most of the problems were connected with fact that FaaS is a very new concept and there are not enough mature frameworks and tools helping in the process of application development. The most critical problem was lack of the support of a local running environment for test purposes of AWS Lambda functions. Amazon does not provide any tool for local testing, only possibility to have multiple stages of Amazon API Gateway at once, where one of the stages can be dedicated to testing while another runs stable version for production. This option is possible but from my experience it slows down the productivity of work just because each change in the code requires uploading new version of AWS Lambda. Even though that Serverless framework has an available third party plugins for testing, from my own experience, the results received from these tools are not exactly the same as in the native AWS running environment.

Even though from transformation to microservices mainly benefits rather large enterprise applications that need to scale, we can say that the transformation to microservices in this smaller example on the Solteq Wellbeing project was successful, where application running solely on AWS Lambda is much more cheaper that on provisioned Amazon EC2 instance.

From my own perspective I find my work very beneficial for my future growth. I found in the Cloud technologies field that interests me and I would like to extend my knowledge further, beyond the scope of this thesis. Especially new cloud services e.g. FaaS needs to become more developed and opens opportunities for programmers willing to contribute to open-source projects such as Serverless framework.

References

Amazon API Gateway. Official AWS documentation. Accessed on 17 April 2017. Retrieved from https://aws.amazon.com/api-gateway/

Amazon DynamoDB. Official AWS documentation. Accessed on 16 April 2017. Retrieved from https://aws.amazon.com/dynamodb/

Amazon DynamoDB: Processing New Items in a DynamoDB Table. Official AWS documentation. Accessed on 29 April 2017. Retrieved from http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.Lambda.Tutorial.html

Amazon DynamoDB: Read Consistency. Official AWS documentation. Accessed on 16 April 2017. Retrieved from http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html

Amazon DynamoDB: Limits. Official AWS documentation. Accessed on 16 April 2017. Retrieved from http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html

Amazon EC2: Pricing. Official AWS documentation. Accessed on 16 April 2017. Retrieved from https://aws.amazon.com/ec2/pricing/

Amazon Machine Images. Official AWS documentation. Accessed on 16 April 2017. Retrieved from http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

Amazon Route 53. Official AWS documentation. Accessed on 15 April 2017. Retrieved from https://aws.amazon.com/route53/

Amazon Route 53 Announces ELB integration, Weighted Round Robin, and General Availability. Official AWS documentation. Accessed on 15 April 2017. Retrieved from https://aws.amazon.com/about-aws/whats-new/2011/05/24/amazon-route53-elb-integration-wrr-ga/

Amazon S3: Configure a Bucket for Website Hosting. Official AWS documentation. Accessed on 22 April 2017. Retrieved from http://docs.aws.amazon.com/AmazonS3/latest/dev/HowDoIWebsiteConfiguration.html

Amazon S3 Product Details. Official AWS documentation. Accessed on 15 April 2017. Retrieved from https://aws.amazon.com/s3/details/

Amazon S3 Storage Classes. Official AWS documentation. Accessed on 15 April 2017. Retrieved from https://aws.amazon.com/s3/storage-classes/

Anicas, M. 2014. *An Introduction to HAProxy and Load Balancing Concepts.* Accessed on 15 April 2017. Retrieved from https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts

AWS CLI: Amazon S3 cp. Official AWS documentation. Accessed on 22 April 2017. Retrieved from http://docs.aws.amazon.com/cli/latest/reference/s3/cp.html

AWS CloudFormation. Official AWS documentation. Accessed on 22 April 2017. Retrieved from https://aws.amazon.com/cloudformation/

AWS Lambda. Official AWS documentation. Accessed on 17 April 2017. Retrieved from https://aws.amazon.com/lambda/

AWS Lambda: Invoking Lambda Function. Official AWS documentation. Accessed on 17 April 2017. Retrieved from http://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-function.html

AWS Lambda: Pricing. Official AWS documentation. Accessed on 17 April 2017. Retrieved from https://aws.amazon.com/lambda/pricing/

Cloud computing. Accessed on 28 March 2017. Retrieved from https://en.wikipedia.org/wiki/Cloud_computing

Chaos Monkey. Official documentation for Chaos Monkey. Accessed on 1 April 2017. Retrieved from https://netflix.github.io/chaosmonkey

Columbus, L. 2016. *15 Top Paying IT Certifications In 2016: AWS Certified Solutions Architect Leads At $125K.* Accessed on 8 April 2017. Retrieved from https://www.forbes.com/sites/louiscolumbus/2016/02/21/15-top-paying-it-certifications-in-2016-aws-certified-solutions-architect-leads-at-125k

Conway, M. E. 1968. *How do committees invent?* Datamation Magazine. Accessed on 29 March 2017. Retrieved from http://www.melconway.com/Home/pdf/committees.pdf

De Santis, S., Florenz, L., Nguyen, D. V., & Rosa, E. 2016. *Evolve the Monolith to Microservices with Java and Node.* Redbooks

Evans, E. 2015. *DDD & Microservices: At Last, Some Boundaries!* Video recording from GOTO Conference in Berlin 2015. Accessed on 1 April 2017. Retrieved from https://www.youtube.com/watch?v=yPvef9R3k-M

Fowler, M. 2004. *Strangler Application.* Accessed on 4 April 2017. Retrieved form https://www.martinfowler.com/bliki/StranglerApplication.html

Fowler, M. 2015. *Monolith First.* Accessed on 1 April 2017. Retrieved from https://martinfowler.com/bliki/MonolithFirst.html

Goldsmith, K. 2015. *Microservices @ Spotify.* Video recording from GOTO Conference in Berlin 2015. Accessed on 1 April 2017. Retrieved from https://www.youtube.com/watch?v=7LGPeBgNFuU

Import and Export API Gateway API with Swagger Definition Files. Official AWS documentation. Accessed on 17 April 2017. Retrieved from http://docs.aws.amazon.com/apigateway/latest/developerguide/create-api-using-import-export-api.html

Introduction to Amazon Simple Storage Service (S3) - Cloud Storage on AWS. Accessed on 15 April 2017. Retrieved from https://www.youtube.com/watch?v=rKpKHulqYOQ

Juneja, V. 2016. *From Monoliths to Microservices: An Architectural Strategy.* Accessed on 29 March 2017. Retrieved from https://thenewstack.io/from-monolith-to-microservices

Kroonenburg, R. 2017. *AWS Cerfified Solutions Architect – Associate 2017.* Online video course - preparation for AWS certification. Accessed on 16 April 2017. Retrieved from https://www.udemy.com/aws-certified-solutions-architect-associate/

Lewis, J., & Fowler, M. 2014. *Microservices a definition of this new architectural term.* Accessed on 29 March 2017. Retrieved from https://martinfowler.com/articles/microservices.html

Lewis, J., & Fowler, M. n.d. *Microservices Resource Guide.* Accessed on 29 March 2017. Retrieved from https://www.martinfowler.com/microservices

Managing Access Permissions to Your Amazon S3 Resources. Official AWS documentation. Accessed on 15 April 2017. Retrieved from http://docs.aws.amazon.com/AmazonS3/latest/dev/s3-access-control.html

Nanoservices, Microservices, Monolith — Serverless architectures by example. Accessed on 17 April 2017. Retrieved from https://medium.com/just-serverless/nanoservices-microservices-monolith-serverless-architectures-by-example-2e95365a0f6f

Panettieri, J. 2017. *Cloud Market Share 2017: Amazon AWS, Microsoft Azure, IBM, Google.* Accessed on 15 April 2017. Retrieved from https://www.channele2e.com/2017/02/09/cloud-market-share-2017-amazon-microsoft-ibm-google/

Ranney, M. 2016. *What I wish I had known before scaling Uber to 1000 services.* Video recording from GOTO Conference in Chicago 2016. Accessed on 28 March 2017. Retrieved from https://www.youtube.com/watch?v=kb-m2fasdDY

Richardson, Ch. 2016. *Refactoring a Monolith into Microservices.* Accessed on 4 April 2017. Retrieved from https://www.nginx.com/blog/refactoring-a-monolith-into-microservices

Serverless framework. Accessed on 22 April 2017. Retrieved form https://serverless.com/framework/

Serverless.yml Reference. Accessed on 22 April 2017. Retrieved form https://serverless.com/framework/docs/providers/aws/guide/serverless.yml/

Smith, C. 2017. *Understanding AWS Lambda Performance—How Much Do Cold Starts Really Matter?* Accessed on 17 April 2017. Retrieved form https://blog.newrelic.com/2017/01/11/aws-lambda-cold-start-optimization/

Subramaniam, V. 2015. *Core Design Principles for Software Developers.* Video recording from Devoxx Conference in Belgium 2015. Accessed on 1 April 2017. Retrieved from https://www.youtube.com/watch?v=llGgO74uXMI

Venkatraman, A. 2011. *In-house vs. moving to the cloud: Data centre learning guide.* Accessed on 15 April 2017. Retrieved from http://www.computerweekly.com/feature/In-house-vs-moving-to-the-cloud-Data-centre-learning-guide

Vladimirskiy, V. 2016. *10 Popular Software as a Service (SaaS) Examples.* Accessed on 15 April 2017. Retrieved from https://getnerdio.com/blogs/10-popular-software-service-examples/