

Master's thesis

Computer Science

From monolith to microservices

Timo Aronen

November 29, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. Valtteri Niemi

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Computer Science

Tekijä — Författare — Author

Timo Aronen

Työn nimi — Arbetets titel — Title

From monolith to microservices

Ohjaajat — Handledare — Supervisors

Prof. Valtteri Niemi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	November 29, 2020	42 pages

Tiivistelmä — Referat — Abstract

Microservice architecture style has become one of the most popular software development architectural styles in the recent years. Microservice architecure is not defining an entirely new independent software architecture, but it is a specific way to implement *service oriented architecture*. Together with the automated software deployment processes it enables organisations to publish new features and fixes to their applications in a rapid and lean way.

Microservice architecture style is usually seen as an alternative way to implement applications compared to the more traditional way where an application, often called a monolith, is a single executable piece of software. Monoliths are thought to be more difficult to develop and maintain, limited in scalability, causing more downtime in production and causing technology lock-ins. Although microservices can resolve some problems with monolithic application, microservices are introducing new different kinds of problems that must be addressed. For example, the amount of deployable components and the complexity of the whole system are increasing.

Many organisations are planning to migrate their existing monolithic applications to a set of microservices. However, it is not clear when this kind of migration is cost-efficient. This thesis is a case study of an organisation which is trying to find alternative for their monolithical style and which has already migrated some applications to microservices. Using the organisation as a case study, we will reflect how some theories from some selected literature are working in practise.

ACM Computing Classification System (CCS)
Computer systems organization → Architectures

Avainsanat — Nyckelord — Keywords

microservices, SOA

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking and Services specialisation line

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Computer Science

Tekijä — Författare — Author

Timo Aronen

Työn nimi — Arbetets titel — Title

From monolith to microservices

Ohjaajat — Handledare — Supervisors

Prof. Valtteri Niemi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	29.11.2020	42 pages

Tiivistelmä — Referat — Abstract

Mikropalveluarkkitehtuurista on viime vuosina tullut yksi suosituimmista ohjelmistokehityksessä käytettävistä arkkitehtuurityyleistä. Mikropalveluarkkitehtuuri ei määrittele kokonaan uutta, itsenäistä arkkitehtuurityyliä, vaan se määrittelee tietyn tavan toteuttaa *palvelukeskeistä arkkitehtuuria*. Mikropalveluarkkitehtuuri yhdessä automatisoidun ohjelmistokehitysprosessin kanssa mahdollistaa organisaatioille ketterän ja nopean tavan julkaista uusia ominaisuuksia sekä parannuksia sovelluksiinsa.

Perinteisessä ohjelmistokehitysarkkitehtuurissa sovellukset koostuvat usein yhdestä suoritettavasta ohjelmasta, jota voidaan kutsua monoliitiksi. Mikropalveluarkkitehtuuri nähdään yleensä vaihtoehtoiseksi tavaksi toteuttaa sovelluksia perinteiseen tapaan verrattuna. Monoliittisten ohjelmistojen ajatellaan olevan vaikeasti kehitettäviä ja ylläpidettäviä, skaalautuvuudeltaan rajoittuneita, aiheuttavan pidempiä tuotantokatkoksia sekä johtavan teknologisiin riippuvuuksiin. Mikropalveluarkkitehtuuri saattaa ratkaista joitain monoliittisten ohjelmistojen yleisiä ongelmia, mutta mikropalveluarkkitehtuuri aiheuttaa myös uusia, erilaisia ongelmia, jotka pitää ottaa huomioon. Yksi huomioon otettava asia on se, että mikropalveluarkkitehtuurissa julkaistavien ohjelmistokomponenttien lukumäärä kasvaa, mikä aiheuttaa lisää monimutkaisuutta kokonaisuuden kannalta.

Monet organisaatiot harkitsevat monoliittiisten ohjelmistojensa arkkitehtuurin vaihtamista mikropalveluarkkitehtuuriin. Ei ole kuitenkaan selvää, missä tapauksissa ohjelmistoarkkitehtuurin vaihtaminen on kustannustehokasta. Tässä opinnäytetyössä perehdytään organisaatioon, joka yrittää löytää vaihtoehtoista tapaa toteuttaa monoliittisia ohjelmistoja ja joka on jo vaihtanut joidenkin monoliittisten sovellustensa arkkitehtuurin mikropalveluarkkitehtuuriin. Työssä tutkitaan, kuinka valitut teoriat kirjallisuudesta toimivat käytännössä kyseisen organisaation kohdalla.

ACM Computing Classification System (CCS)
Computer systems organization → Architectures

Avainsanat — Nyckelord — Keywords

microservices, SOA

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking and Services specialisation line

Contents

1 Introduction **1**

2 Microservice architecture **3**

2.1 Relationship to Service Oriented Architecture 3

2.2 Challenges when adopting MSA 7

3 Deployment pipeline **9**

3.1 Continuous integration 9

3.2 Artifact repository 10

3.3 Automated testing 11

3.4 Unit tests 12

3.5 Integration tests 14

3.6 Continuous delivery 17

3.7 End-to-end tests 17

3.8 User interface tests 18

3.9 Continuous deployment 18

4 Development and Operations **20**

5 Challenges with migrating from monolith to microservices **22**

6 Case Business Finland **25**

6.1 Funding process 25

6.2 Service oriented architecture in funding process applications 26

6.3 Microservices in funding process applications 29

6.4 DevOps evolution in Business Finland 34

7 Conclusions **36**

References 39

1 Introduction

Microservice architecture (MSA) style has become one of the most popular software development architectural styles in the recent years. Together with the automated software development and operation processes it enables organisations to publish new features and improvements to their applications in a rapid and lean way.

Cloud computing is a computing model where the necessary computing resources are accessed over a network [28]. The resources can be, for example, networks, servers, storage, applications and services. The resource management and allocation is dynamic and scalable. The resources are scaled according to the client need using minimal effort. At any given time the client for the cloud computing services can expect to have as much resources available as needed. It is even possible that the client of the cloud computing services is not self hosting or maintaining any computing resources at all. Instead the client purchases all necessary computing resources from some cloud computing service provider.

Many organisations are seeing cloud computing as a cost-efficient alternative compared to the hosted servers. Microservices are often said to be *cloud-native* which means they are designed according to cloud-focused design patterns [32]. This design allows organisations to deploy their MSA applications relatively easily to cloud environment instead of executing them in hosted servers.

Microservice architecture style is usually seen as an alternative way to implement applications compared to the more traditional way where an application, often called a *monolith*, is a single executable piece of software. Compared to MSA applications, monoliths are thought to be more difficult to develop and maintain, limited in scalability, causing more downtime in production and causing technology and vendor lock-ins.

Developing an application using MSA style is more expensive than implementing the application using monolithic style. However, during the maintenance phase of the application, MSA style is expected to be less expensive compared to monolithic style. In addition, MSA offers some solutions to other problems commonly encountered with monoliths.

Some organisations are having maintenance and scalability issues with their old monoliths and they are seeing MSA as a potential solution to these issues. Although MSA seems to solve some problems with monolithic applications, it is not a silver bullet, and not

applicable to every case. There are no clear paths how to migrate an existing monolith to a MSA application. In some cases the migration from monolith to MSA would not be cost-efficient even in a long term.

The automatisation of various phases in the development of a MSA application is essential. Implementing an application in MSA style causes a drastic increase in the amount of deployable components. In addition to the increased amount of the components, the dependencies between the components are creating complexity that would be difficult to manage without automatisation.

This thesis is a case-study about an organisation that has decided to migrate their architecture away from monolith systems. The organisation is evaluating different approaches for new architectural style and MSA might solve some of the problems they have with monoliths. At first we will introduce the MSA and background for the migration in this particular case. We will reflect the solutions found from the literature to the issues in our case. We try to find out what are the prerequisites needed when adopting microservice architecture and whether it is cost-efficient for the case-study organisation to migrate existing monolithic application to a set of microservices.

2 Microservice architecture

At the time of writing MSA have existed about 10 years [16]. Microservice architecture style started to appear in the industry at the beginning of 2010 and later the academia started to also research this new style. The number of publications related to microservices has grown drastically since 2015 [36] and there was a lot of research work going on the topic in 2019 [4].

One of the first and often cited descriptions about MSA style is written by Lewis and Fowler [16] in 2014. According to them MSA application consists of multiple small-sized services (*microservices*) which can be developed, tested and deployed independently. Microservices are communicating with each over the network using lightweight mechanisms. Each of the service in the system can be implemented using any programming language. Wampler and Clark are defining this kind of practice as *polyglot programming* [46].

Netflix and *Amazon* were among the first organisations to adopt MSA style even though they did not use that name for their architectures since the term did not exist yet. For example, Netflix used a term *fine grained service oriented architecture* to describe their architecture that was resembling MSA. The term *microservice* was discussed at a workshop of software architects in 2011 and in another workshop in 2012 the architects decided to use the term for this new architectural style [16].

2.1 Relationship to Service Oriented Architecture

In this section we will reflect MSA on *service oriented architecture* (SOA) [27]. It is helpful if the reader knows some basics about SOA, but that is not necessary. We are comparing some aspects between MSA and SOA and by this way introducing the ideas behind MSA.

There are a lot of patterns that MSA and SOA are sharing. There is also some debate about what is actual relationship between MSA and SOA [16, 49, 4]. Both of these architectural styles are service oriented, but SOA has been existing much longer than MSA. Some see SOA as a legacy architecture and MSA as a modern variant of it. Some call MSA as ”SOA done right” [49]. There are clearly some differences between MSA and SOA, but according to Bogner et al. [5] MSA and SOA are sharing a large amount of

SOA patterns and they have only some distinct areas of small differences. Therefore MSA style should be seen as ”a specific development and deployment approach for SOA”.

Growing number of organisations are implementing and some are migrating their systems using MSA. According to Alshuqayran et al. MSA was the major choice for architectural style in the service oriented software industry in 2016 [1].

One of the differences between MSA and SOA is the granularity of a service. The size of the service in MSA is required to be minimal and to focus on one small thing only. Such requirements do not exist in the SOA world. This can make a big difference in the design and the implementation of the system. For example, in SOA we might have a considerably large customer service, which would take care of all the system activities related to the customer. When there would be a need to create a new customer into system, a client of the customer service would dispatch a 'create a new customer' request to the customer service. The client would provide the customer related data as request parameters to the customer service. The customer service would use the provided parameters and perform all the necessary system operations related to the customer creation. At the first, the service performs a credit check on the customer. After the credit check, the service sends the information of the customer to a third party company for sending a welcome gift to the customer. At the last step, the service sends a confirmation email and then persist the customer data into a database (see Figure 2.1).

In the previous example, all the business logic related to the customer creation is implemented in the customer service. Using MSA the business logic would be spread into multiple microservices. In MSA there is a separate service for each task. Figure 2.2 is an example of similar system, but implemented using event based MSA. In event based system, the functionality is triggered by events occurring in different phases of the proccesses. It can be implemented so that each microservice registers to a *message broker* and subscribes the events it wants to be notified about. Now, the new customer would be created by submitting a 'customer created' event to the message broker. Message broker would then notify all the microservices, which have subscribed the event.

Another difference between the SOA and the MSA is that there is no *enterprise service bus* (ESB) in MSA. In contrast to MSA, in a SOA there is an ESB which routes and transforms the service requests and responses. There exist multiple ESB implementations and depending of the vendor they are providing different set of mechanisms for message transformation, routing and handling. In some ESB implementations it is even possible to execute process choreography and service orchestration. Using an ESB it is possible to

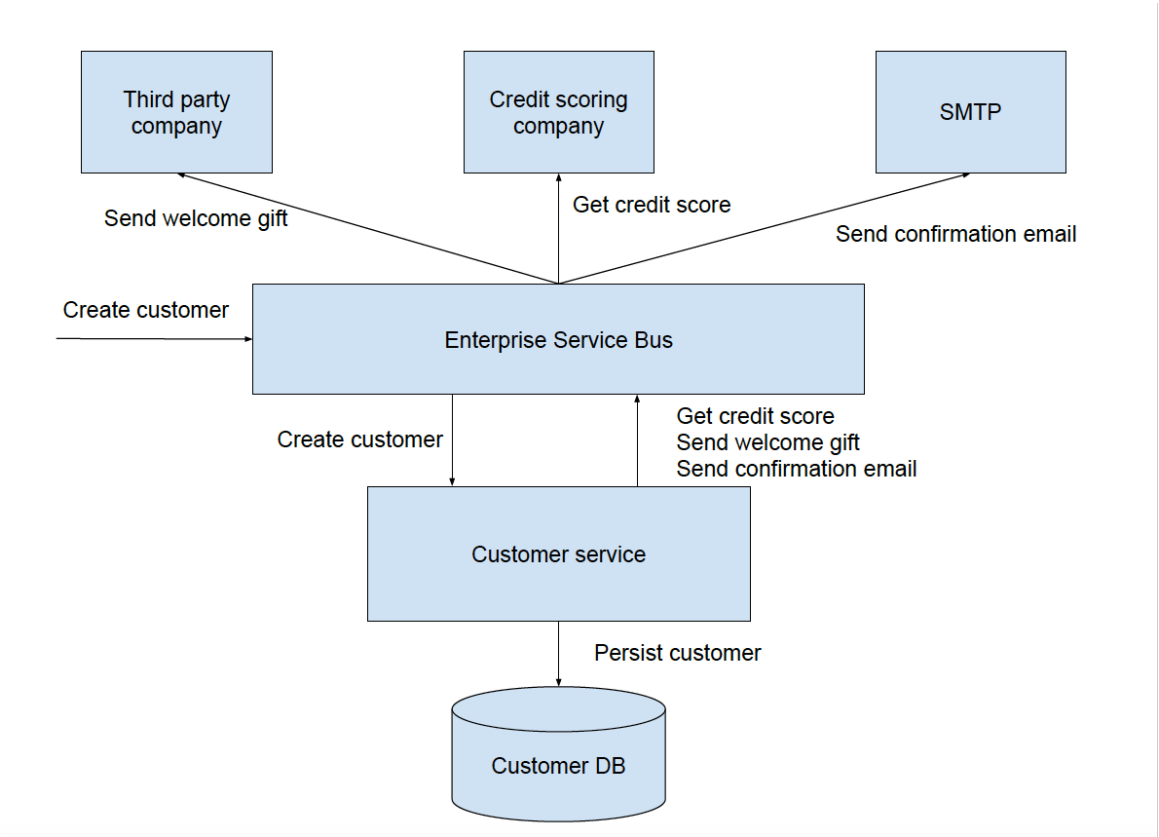


Figure 2.1: Customer creation in SOA

manipulate service request and response of the routed messages. In addition the routed messages could trigger additional service calls. It is also possible to implement some other business logic based on the messages or the results of the additional service calls. A term *smart pipe* is used to denote this kind situation, when the message is not only transferred by the service bus, but some additional functionality is executed as well. If a smart pipe is shared between services, there will be a strong dependency between the services. The smarter the ESB is the dumber the services are. This kind of architecture is totally different from what is recommended for MSA. Instead of smart pipes, MSA prefers ”choreography over orchestration and dumb middleware, with smart endpoints” [34].

One of the ideas behind the ESB is to implement services and integration using configuration instead of adding new software code. Unfortunately each of the ESB vendors tend to have their own tool for configuring their ESB and there is no common configuration language. If there is a lot of logic implemented in the ESB, it can lead to a situation where a change of an ESB vendor will be expensive, since all the configuration must be reimplemented. The developers are also sometimes finding it hard to configure and implement

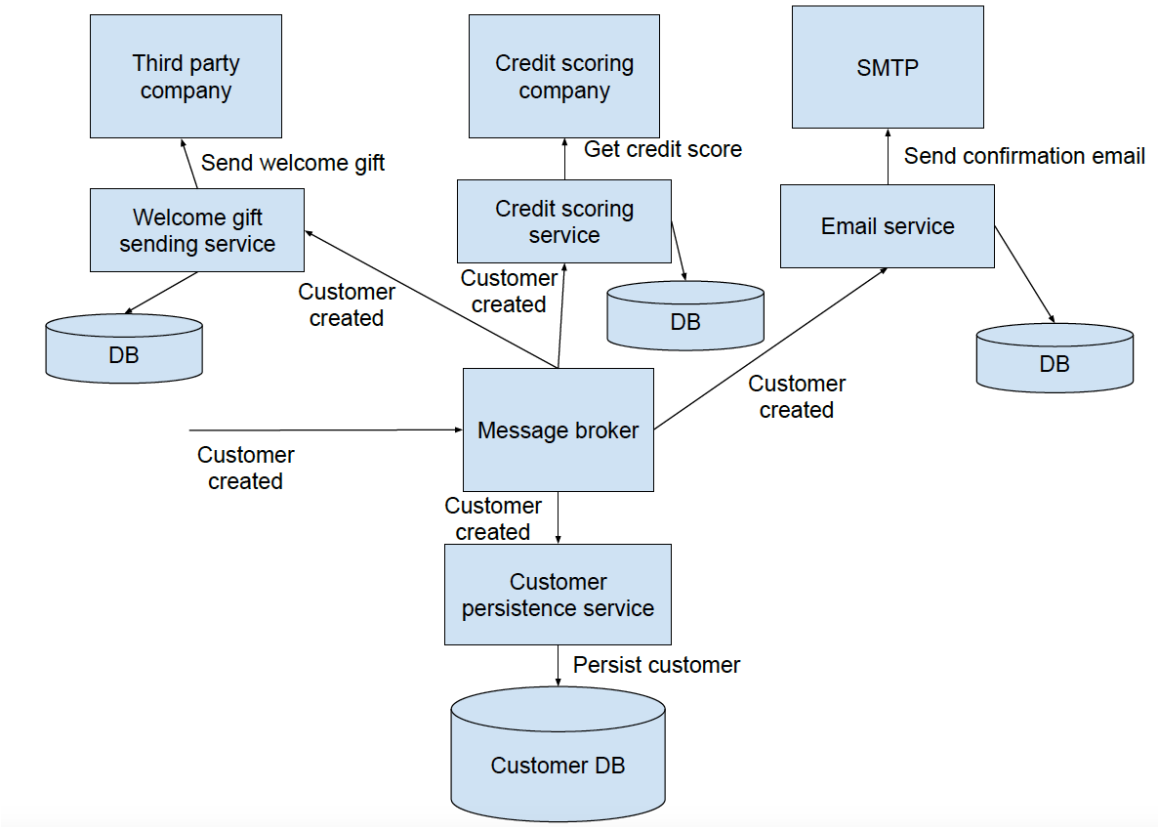


Figure 2.2: Customer creation in MSA

services using an ESB. This kind of situation is also known as *vendor lock-in*.

A microservice is a small application by itself. It is doing some well-defined small job, usually as a part of a bigger application (MSA application). A microservice could be, for example, an application that sends emails or validates files against viruses. An application implemented using MSA is consisting of multiple, small, independent, loosely coupled services. Microservice architecture promises to solve problems related to scalability, maintainability and technology lock-ins for an application. But like in all architectural styles, there are some trade offs to be made when choosing MSA for an application [34, 44].

The first part of the word *microservice* is usually thought as the code base of the service implementation should be really small. The actual size of the code base of a microservice is not defined. Sometimes it is hard to determine the right granularity for the service. In practice the size of a microservice is small enough, when a team developing the service decides so. In general a microservice should be small enough for one small team to develop and maintain it [16, 34]. The optimal size for the team is often defined using ”two pizza tea” rule introduced by Jeff Bezos, the CEO of Amazon [11]. The rule means that the

team developing the service should not be bigger than two pizzas can feed. Usually this is interpreted to mean 6 to 8 people.

When the code base for a service is small, the function of the service is easier to understand. Microservice architecture follows also *single responsibility principle*, which means that each service is focused on doing one thing well [34]. By following this principle the service will be simple to understand and easy to maintain. If necessary, the whole implementation of the service could be rewritten in some sensible time. Although a single microservice could be easy to understand and maintain, implementing an application using multiple microservices increases the complexity of the whole application, since the application will be distributed.

Independence and the term *loosely coupled* means that the services are separate entities, which are deployed in separate units and can be updated without changes to the other services. Usually the services are exchanging data using some commonly know format over network, typically using JSON over HTTP [34]. That relieves us for using some programming language depended shared libraries between clients and servers. The JSON-interface of the service should be also designed in a way that there would not be any tight coupling made to the interface in the implementation of the client or the service. This allows each team developing a microservice to freely choose the implementation technology most suitable for them.

2.2 Challenges when adopting MSA

Although MSA is currently one of the most adopted style for developing new applications, there is relatively small amount of research made of it. It is hard to measure what benefits MSA could offer over a monolithical application and there is no clear way how to migrate from different existing architectures to MSA.

One of the basic ideas behind MSA is to share as little as possible and support polyglot programming. This means that, in the worst case, we need to implement and maintain the same exact logic in multiple applications. Or course if the common logic is complex and some domain specific, it should be extracted to a separate microservice. However, sometimes there can be validations or database access implementations, which will be inefficient when used as a service. For example, the validation of the syntax of *value added tax code* (VAT) or *social security number* (SSN) could be done using some validation service. But when we are processing a huge batch containing multiple VAT or SSN,

validating each value using a service could create a performance problems. If we would like to avoid the service calls, we would have to implement the same exact logic everywhere. Although MSA discourages the use of shared libraries, this kind of code could still be shared as a library. However, creating a library requires careful design that sharing the library will not create unnecessary dependencies between the services. Although, if polyglot programming is employed, there might be a need to implement the same library in a different programming language.

3 Deployment pipeline

The process where an application is built from the source code and deployed to production can be referred as *deployment pipeline* [19]. The process can be fully automated or it can include manual steps. Figure 3.1 illustrates an example of a deployment pipeline. The process starts from 'Development environment' where the software is implemented by the developer(s). When a developer is satisfied with the implementation he or she will commit changes to the source code to a *version control system* (VCS). When there is a need for multiple developers to edit the same source code, some kind of source code management is needed. Otherwise it would be hard for the developers to merge their simultaneously made commits. There exist multiple commercials and open source-tools, such as *ClearCase* [20], *Git* [18] and *Mercurial* [29], which can help developers to solve problems if there are conflicts when the source code is merged. The other steps of the example deployment pipeline will be examined in the following sections.

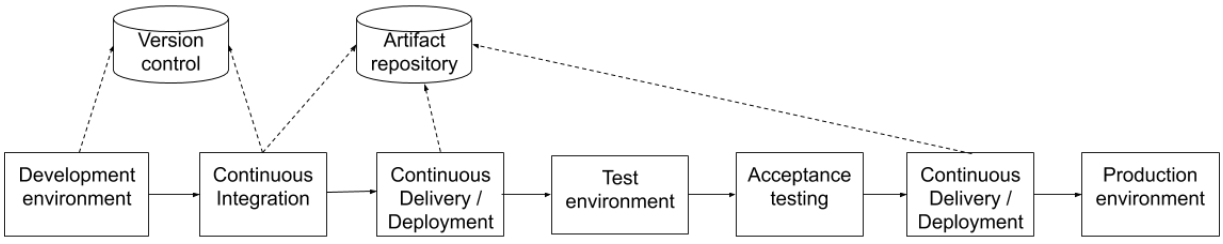


Figure 3.1: An example of a deployment pipeline.

3.1 Continuous integration

Continuous integration (CI) is a software development practice where software under development is continuously build and tested [14]. Although it is possible to implement CI manually, there exist open-source and commercial CI servers available for that as well. A common practice to implement CI is to setup on-premise or cloud based CI server. When source code is changed in VCS, CI server automatically pulls the changes and executes necessary tasks for building and testing the software. If there are problems in the build or if the tests are failing, CI server is providing immediate feedback to the developer. Often CI server is also performing statistical code analysis, vulnerability scan and other tasks

measuring the quality of the software.

Building and integrating software continuously helps to detect problems early and make them easier to resolve. It is easier to resolve a problem in a software immediately when it appears instead of when it fails in production. If a developer commits problematic code changes to version control, in the best scenario the build is failing in CI server only after some seconds or minutes after the commit. The developer will be notified of the problems and he or she will revert the code changes or fix the problem immediately. This will reduce the amount of problems ending up to the production and will make it easier to debug other problems it they appear. The longer a problem exists in a software the harder it is to resolve it [14].

Continuous integration server will always detect compilation problems, but other problems will be found only if there are tests covering problematic parts of the code. Usually tests are automated and CI server executes those [34].

3.2 Artifact repository

The phases during a software development process are producing different artifacts. Usually at the beginning of the process some requirement and design documents are produced. Later the software will be implemented in the form of source code and implementation documentation. Any kind of output from any phase of the software development process can be considered as an arfifact.

When a source code of a software is compiled and packaged, a binary file will be produced. This produced artifact can be an executable application or a shared library. An executable application is an artifact that can be copied to a suitable operating system and executed by the user. A shared library is a compiled binary files containing logic that applications are sharing when they are executed.

Artifact repository is a tool for storing, versioning and managing artifacts. It allows users and systems to store and retrieve a specific version of an artifact. In our example, see Figure 3.1, deployment pipeline the CI server is packaging the software to an artifact and deploying it to an artifact repository. From there it can be later retrieved and installed to test and production environments.

3.3 Automated testing

Software testing can be done manually or it can be done automatically. If tests are executed automatically by a CI server we can say that the software is automatically tested [12]. According to a case study by Karhu et al. major benefits of automated testing over manual testing includes improvement in the software quality and reducing the time needed for testing the software [26].

Automated tests can be divided into three layers as Cohn explains in his book [8]. Cohn introduces the *test automation pyramid*, where at the lowest and broadest level are the unit tests, in the middle level are the service tests and at the top are the user interface tests. The idea of this visualization is to show different aspects related to automated tests at each layer.

User interface tests are at the top because they are usually the most expensive to create and maintain. However, when our build passes the user interface tests we can be confident that our software is working as expected [34].

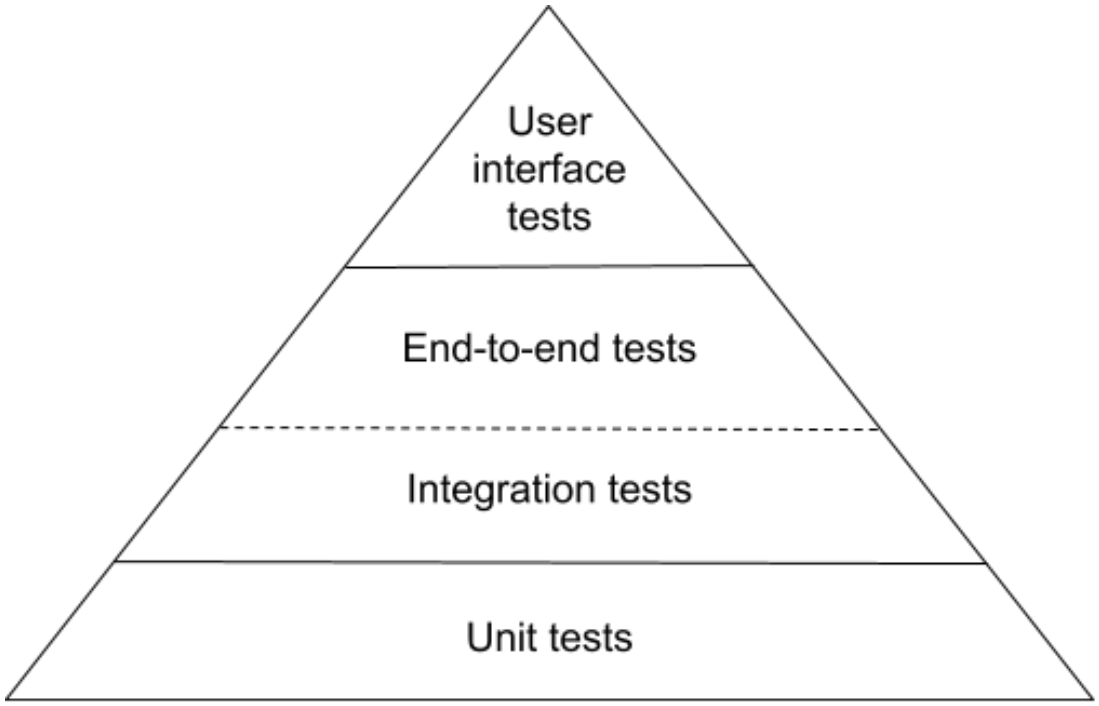


Figure 3.2: Adapted version of test automation pyramid by Cohn [8].

Unit tests are at the bottom of the test automation pyramid, because they are usually the most inexpensive to write and maintain. However, the unit tests are testing only isolated modules in the software. Consequently, when our build is passing the unit tests

we are verifying only that the tested modules in our software are working as expected. The business processes in the software can involve execution of multiple modules. Thus, even if our build passes the unit tests, we can not be confident that all business processes are working as required.

Service tests are at the middle of the test automation pyramid. They give us more confidence than unit tests but they are more expensive to write and maintain. Service tests are testing the interaction between the modules and we can use service tests to verify the business processes.

Because the unit tests are fast to execute and they can provide fast feedback to the developer, they should be executed before time consuming tests at the top of the pyramid.

Test automation pyramid by Cohn is referenced in many books and research papers and many authors have adopted slightly modified versions of it. Grouping tests into different categories can be done in multiple ways and as Newman explains, people tend to have different opinions what service and unit tests actually are [34]. In this thesis test automation pyramid is also altered, service layer testing is divided into two new groups: end-to-end (E2E) and integration tests (IT) as shown in Figure 3.2. At the abstract level they can be thought as service tests.

End-to-end tests are testing services with actual integrations to real services while integration tests are testing module and service integrations where the actual implementations are replaced with some *test doubles*. Usually it is necessary to write service or module implementations for testing purposes only. They are often called mock, stub or dummy implementations. In his book [30] Meszaros describes five such esting implementations and uses a generic term test doubles for all of them.

User interface tests can be considered also as E2E tests, but we make a separation here, since UI tests are often testing the system through a browser.

3.4 Unit tests

Unit tests are typically testing a single function or a single unit of a software [40] and according to Cohn they are usually implemented in the same language as the tested software [8]. The purpose of a unit test is to verify that the tested fraction of a software code is behaving according to its specification. For example, the following Java function is calculating the total value of an item using tax free price of the item and tax percent

as input parameters:

```
public final class Calculator {
    ...
    public static double totalAmount(double price, double taxPercent) {
        if (taxPercent > 24) throw new IllegalArgumentException("Tax percent
            can't be over 24");
        return price + price * taxPercent / 100;
    }
    ...
}
```

In the function named `totalAmount` there is an assert that checks the `taxPercent` named argument has a value less than 24. We could also write additional asserts for the input arguments of the function. For example, we could also verify that the value of the `taxPercent` argument is greater than 0 and so on. For testing the function we can implement the following unit test:

```
class CalculatorTest {
    ...
    @Test
    public void test_totalAmount() {
        assertEquals(5.58, Calculator.totalAmount(4.5, 24));
    }

    @Test
    public void test_totalAmount_too_high_tax_percent() {
        assertThrows(IllegalArgumentException.class, () -> {
            Calculator.totalAmount(4.5, 25);
        });
    }
    ...
}
```

The unit test above includes `test_totalAmount` named test that is verifying the return value of the `totalAmount` function using pre-defined values as function input arguments. The other, `test_totalAmount_too_high_tax_percent` named test, is verifying that the function is reporting an error for an invalid argument. The more unit tests there are, the more we can assure to ourself that the code is working as required. The idea is that when the implementation of the function `totalAmount` changes, our test will verify that the software is still working as expected.

The chosen software development methodology usually defines when and who composes unit tests. Sometimes the test is written after the implementation, but in Test-Driven Development style (TDD) the test is written before the implementation [22]. Tests can be composed by dedicated tester or team of testers, but usually unit tests are written by developer or development team [40]. Sometimes it is even possible to use tools that can generate all or some parts of required unit test code.

A unit test verifies only some fraction of the whole software and it is often necessary to program testing specific drivers or stubs. Since the purpose of a unit test is to test only some small part of an application, we need a driver for executing only that part of the code we want to test. If the code under the test is trying to integrate with other modules or some external services, we need to somehow stub or mock the integration. The stub or mock is an implementation that we have programmed for testing purposes only and it simulates the actual integration. The reason for doing this is that we want to test only the logic in some small unit [47]. We do not want to test the implementations of the integrations. Unit tests are also supposed to be fast and if we would be retrieving data from database instead of mocking it, the execution of the test would take much more time. The execution of a single unit test is supposed to be really fast and thousands of them could be executed in minutes [34].

There are several testing frameworks for different languages helping programmers to write automated unit tests. One of the most know is JUnit [25] for Java language. JUnit has inspired developers to create similar frameworks to other programming languages as well and they are generally called XUnit frameworks [22, 17].

3.5 Integration tests

Integration tests are testing the actual integrations in the software. According to Sam Newman, service layer in the automated testing pyramid is a bit confusing because the

term 'service' is so broadly used and there are many interpretation of what service level testing really means [34]. Newman describes that service tests are bypassing user interface and testing the services directly and usually they are larger and slower than unit tests. In most of the XUnit frameworks these kind of tests are known as integration tests and we can consider them as service tests in the automated test pyramid.

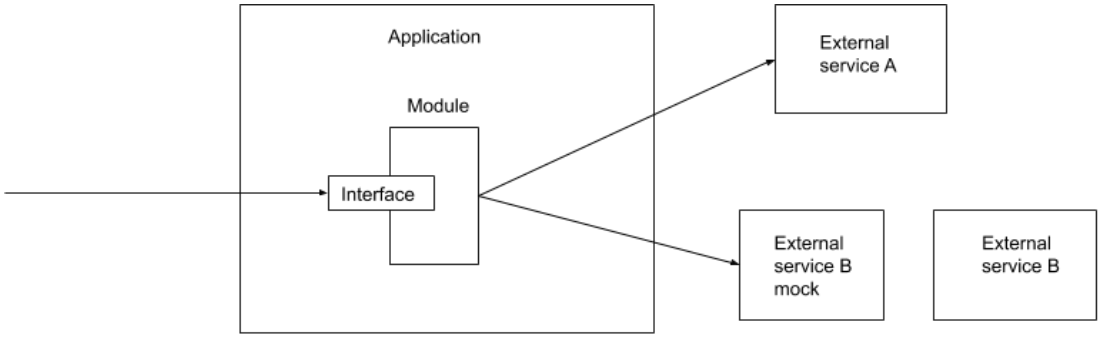


Figure 3.3: Mocking external service in integration test.

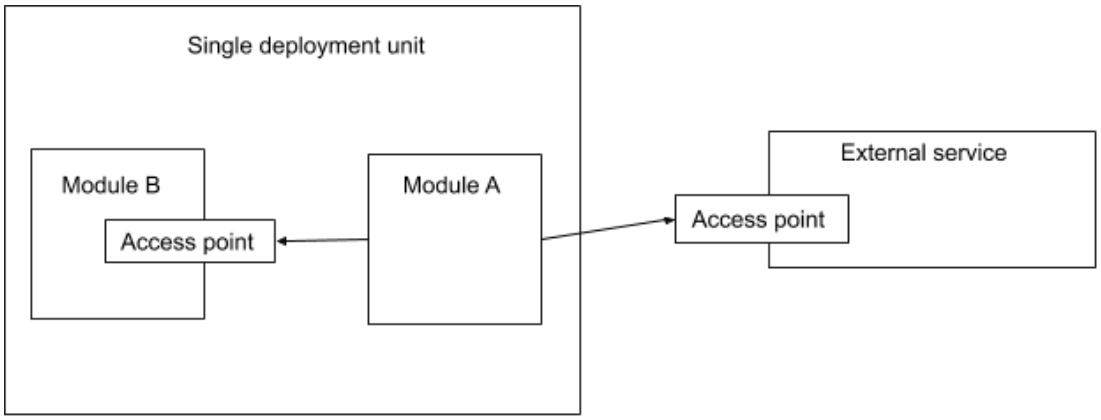


Figure 3.4: Illustration of different integrations.

Integrations in the software can be seen as integrations with modules or libraries in the same deployment unit or as integrations with external services or as integrations with some kind of databases. Figure 3.4 is illustrating some of different types of integrations. In the Figure module 'Module A' has an internal integration with a module 'Module B' and an external integration with a service 'External service'. 'Module A' has also an integration with a software library 'Library' that is bundled to the same deployment unit.

'Library' could also be located in host operating system instead of deployment unit. Testing an internal integration or integration with a software library is much easier than testing an integration with an external service. Since internal integration is in the same deployment unit, modules can be deployed and tested together. The same is true also with the

library, if it is bundled into the same deployment unit. If the library is located in the host operating system we have to take care that the version of the library is compatible with our deployment unit.

At the abstract level testing an integration with an external service or a database can be considered similar. Usually a working network connection and an access to the external service is needed. Only if the external service is executed in the same operating system and the service can be accessed via socket, the network connection might not be needed. When we are testing an external integration we have to make sure our service (or database schema) is compatible with our deployment unit. There can be problems if we want to test two different versions of the deployment unit and they require different, incompatible, versions of the external service (or database schema). Somehow we need to take care that our deployment unit is tested against correct version of the external service. When an integration to an external service fails it is usually much harder to debug the reason for the failure. The reason might be, for example, that the external service is failing or there is a network or an authentication issue. For debugging the failure, we need to gather log files and other information from multiple sources, analyse them and try to determine the root cause for the failure.

Sometimes we want to test only some of all the possible integrations. If this is the case, we might need to mock or stub the other integrations to be able to execute the integration test. Figure 3.3 is illustrating this kind of situation. Here we are testing an integration to an external service 'External service A' and our software needs to call an external service 'External service B' to get the input values for the service under test. If, for some reason, we are not able to or we do not want to make the actual call to service 'External service B' we need to somehow intercept the call and provide the values to our software as they would come from 'External service B'. There are many ways to do this. This kind of interceptor can be written by the developer, but there are also different kinds of testing frameworks for different programming languages for making it easy to mock or stub the actual call. When we are mocking an integration, the test is describing what happens when mocked integration is called. When we are stubbing an integration, we are replacing the actual integration with our own implementation which is emulating the integration and usually works only with some fixed input values.

In our example deployment pipeline, see Figure 3.1, CI server is deploying the artifcat to an arfitact repository after it has successfully passed unit and integration tests. From there the arfifact can be later retrieved and installed to different execution environments.

3.6 Continuous delivery

To be able to execute an application, the application needs to be installed into some operating system. Usually application is installed by copying a binary file to an executing environment. Sometimes the application also needs a configuration file in the execution environment and that must be copied as well. In addition, there might be some other tasks that must be done before the application is available to the end user.

Application installation and related tasks can be done manually or automatically. In the example pipeline see Figure 3.1, the artifact is automatically retrieved from the artifact repository and then it is automatically installed to an execution environment. This kind of practise where a software is always kept in a stage that it can be delivered to the end users as fast as possible is known as *continuous delivery* (CDE)[33, 15].

3.7 End-to-end tests

End-to-end (E2E) tests are verifying the whole system functionality after it is deployed. User interface tests are also E2E tests, but we will introduce them separately in section 3.8, since they are more complex than E2E tests without the UI layer.

When our system is passing the E2E tests we can be more confident that our system is working as expected compared to just passing the unit and integration tests. However, E2E are more expensive to write and maintain than unit or integration tests so it might not be cost-efficient to try to test all the functions of the system using E2E tests. In unit and integration testing we are testing isolated part of software using mocks and stubs, but in E2E testing we are verifying system functionality with actual integrations and that brings some challenges for testing and debugging.

The first problem is that we need to deploy the software to some production-like environment and our automated E2E tests need to be able to communicate with necessary access points. Often we need to initialise the system to some suitable stage before testing, so we need suitable tools for that purpose. For example, we might need to set some values for data in database before the test is executed.

Once the software is deployed and initialised to suitable stage we can run E2E tests on it. If we have failures in our tests, the next problem is to find the root cause for the problem. A test can be failing because of an issue in our system or it can fail if we have

not initialised the system in the appropriate way. The test can fail also because of some external reason. For example, our system might use some third party service and fail because the third party service is failing. Or it might fail because of some network or authentication problems. In any case, because of the complexity, it is much harder to find out the reason for the test failure in E2E test compared to test failures in unit or integration tests.

3.8 User interface tests

User interface (UI) tests are testing the actual interface for the end user. They are verifying that the UI of the application is behaving as expected. For example, if there is a feedback form with a text box and a submit button, UI test could verify that they are visible and enabled when a user is requested to give feedback. Moreover, UI test can verify that the application is working as defined when a feedback form is submitted. If the feedback is persisted in a database, the test would retrieve the submitted feedback from the database and verify it.

It is also possible to verify the behaviour of UI using unit testing by isolating the UI from other components using test doubles. On the other hand, when an UI test is verifying that user input is persisted we can say that it is also an E2E test. To be able to execute UI test as E2E test, we need to have a suitable driver for running UI tests. We need the driver for the test, because we need to program a test that can interact with the UI. There exist several commercial and open-source tools, including *Selenium* [41], *Robot Framework* [38] and *SilkTest* [42] just to name a few that can be used for automated UI testing.

When an application is passing an UI test that is also an E2E test, we can be quite confident that the application is working as expected. However, configuring an additional tool and maintaining UI tests is more complex than testing applications access points using other types of E2E tests. For these reasons UI tests are at the top level in test automation pyramid.

3.9 Continuous deployment

At the last step of the deployment pipeline, the software is made available to users. Often this means that the software is installed to a production server. The installation can

be done manually or automatically. According to Humble and Farley [19], the practice where the deployment to the production is made automatically is know as *continuous deployment* (CD). However, most of the literature does not make a distinction between continuous deployment and continuous delivery and the terms are used interchangeably [39].

There are number of benefits that CD can offer. New features and improvements can be delivered to the users in lean and rapid way. This may increase the customer satisfaction. The users can also identify and report about new problems, which are usually easier to resolve immediately than after long period of time. However, CD might require investments to other parts of deployment pipeline or to the way how organisation is operating as Rodrigues et al. have found in their systematic mapping study [39]. For example, quality assurance stages in the deployment pipeline might need some additional effort.

4 Development and Operations

Development and Operations (DevOps) is an acronym for processes and methods which aim to enable rapid software deployment. In their article R. Jabbari et. al [21] suggested the following definition for DevOps: "DevOps is a development methodology aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices".

According to Vogels, Amazon is following "you build it, you run it" philosophy [35]. This means that the team who implemented a service is responsible for maintaining it for the whole lifecycle of the service. Following this philosophy the maintainability of the service can improve [31]. The philosophy is also supported by Newman, who defines in his book that in MSA the team has the *service ownership* [34]. This means that the team maintaining the service is implementing all necessary code changes to the service. According to Newman, in many cases the ownership covers everything from creating the source code to building, deploying and maintaining the service.

In MSA the amount of deployable artifacts increases adding complexity to installations. On the other hand, microservices are supposed to support CI and CDE [31] which means that microservices should be installed automatically at least into integration test environment. Installation automation can be achieved by utilising different DevOps practices.

Some companies are deploying software changes automatically to production. After successful build and if the tests are not failing and the quality requirements are filled, CI server or some other tool is deploying new version of software automatically into production server. Deployment can be done using some provisioning tool like Ansible [2], Puppet [37] or Chef [7].

Microservice architecture can be implemented without DevOps and DevOps can be implemented without MSA. However, according to Google Trends the popularity of these two concepts have been growing at equal rate after 2014 [3]. When a monolithic application is migrated to MSA the amount of deployable units increases. When a new application is implemented using MSA the nature of MSA means that there will be multiple deployable units. The growing amount of deployable units in MSA creates complexity. However, automatisation can be used to manage the increasing complexity [34, 3]. High level of

automatisation in all levels of MSA development and deployment pipeline can be reach by implementing DevOps practices. This may explain why the popularity of MSA and DevOps have been following the same growth path since 2014.

Implementing and maintaining a deployment pipeline requires investments that are sometimes hard to justify. When there is a budget for improving some part of an application it is often forgotten that sometimes deployment pipeline requires changes as well. Also implementing a fully automised deployment pipeline from the scratch comes requires investment without immediate return of it. When a company has only a few applications which are deployed only a couple of times in a year, it might not be cost efficient to invest deployment pipeline. However, when there are tens or hundreds of microservices to be managed and deployed, there is a need for efficient deployment pipeline.

5 Challenges with migrating from monolith to microservices

One of the biggest challenges, when migrating from a monolith to a set of microservices is the complexity related to changing a single deployable monolith application to a distributed application. This kind of migration introduces many problems to be solved and it should be really carefully thought, if the migration would be worth of doing.

Developing and deploying a monolithic application is usually straightforward and simple in the beginning. The application code resides in one repository and all the logic is there. The software is compiled, tested and packaged into single binary unit which is then transferred into a production server and executed there. When the same application is implemented in MSA style there will be much more complexity related to testing and deployment of the application. Also more attention should be paid on the design of the whole system and service interfaces.

There is no overhead in system resource consumption when monolithic application is executed in the server. When the same application is implemented and executed as MSA application, there is a need to execute the application in multiple processes. Execution of multiple processes is requiring more memory and low level system resources. The distributed nature and the communication over network in MSA application is setting more requirements for the network than a monolithic application. If there is a need to execute all MSA services in the development environment, also the development environment is required to have more resources than in monolithic application development. However, often when developin a MSA application there is a need to develop only some isolated service of the application. Consequently the requirements for the development environment are usually less demanding compared to monolithic application development.

In a MSA application the business logic is distributed into multiple services which can make it hard to understand how a certain business process is implemented and what services are employed for that. Although, services in MSA are independent deployable units, sometimes services are depended on some new features provided by some other service. This introduces complexity related to the versioning of the services and deployment of the services. If related services are not deployed at the same time into production server, the

service providing new features must be deployed first. If the new feature of the service is not compatible with the old application, the service must support also the old version of the feature.

Testing a MSA application is more challenging compared to a monolithic application. Unit and integration testing of a monolithic application is simple, since all the required code can reside in one repository. The code is retrieved from the repository and the application is compiled and tested. This can be done even with a simple build script without any need for a continuous integration (CI) server [14] or other DevOps tools. This is not the case with MSA which relies heavily on CI and DevOps. Without proper CI and DevOps MSA will loose some of its biggest benefits.

The more services a MSA application consists of the more CI is needed. Different services can be developed by different teams and CI is a method that tries to guarantee that the MSA application is still working after changes in the services it depends on. When a developer makes a change to the source code, CI server detects the change and automatically builds and tests the application. This can be done automatically if there are suitable DevOps tools available. Monolithic application development can also benefit from using CI and DevOps, but they are not so important than in MSA development.

Testing a MSA application is difficult, though testing a single microservice is easy. Because a MSA application is distributed, testing a business process implementation is hard. A business process can consists of multiple microservice calls. Testing a MSA application requires that all necessary microservices are deployed and their versions are correct. When a business process test is executed we need to verify that all required microservices are executed as well. We need to also verify that each microservice did what is was supposed to do. Testing a single microservice or a monolithic application is much easier, since there is only one place to look for errors.

If a monolithic application does not need any external services there is no need for a network, since all the needed logic could be executed in a single server. Application implemented using MSA basically always needs a network, since the services are usually communicating with each other using some lightweight mechanism like JSON over http. The communication between processes using a network is always slower than communicating inside a single server, so latency must be taken account when implementing the application.

The distributed nature of a MSA application indroduces new problems related to security. Since in the MSA there are more access points open in the network, the potential attacker

has wider surface to attack. If the network communication involves sensitive information or if we need access control to service interfaces, there will be also security issues which must be addressed.

In addition, because of the distributed nature, there will be more processes to monitor. When a monolithic application is exectuted the information needed for the monitoring can be accessed in a single server. When a MSA application is executed, the monitoring is more complex. The information needed for monitoring is also distributed and gathering it for reporting can be challening.

The initial costs for MSA based application are higher than with monolithic application. However, depending of the lifetime of a monolithic system the migration from a monolith to a MSA application could be cost-efficient, since the maintenance costs could be lower in MSA in the long run. Based on interviews in empirical investigation by Taibi et al [45] the return of the investment in migration is less than two years in 33% of the cases and less than 3 years in the rest of the cases.

6 Case Business Finland

Business Finland (BF) "is the Finnish government organisation for innovation funding and trade, travel and investment promotion" [6]. Business Finland was formed in 2018 when two government organisations, *Finpro* and *Tekes*, were merged to one [13]. In practise BF is offering different kinds of services for the organisations in Finland with the purpose of improving their business capabilities. These services include financial support, business networking and business coaching.

Among other duties, BF is offering project funding to the companies and research organisations registered in Finland. Funding can be gratuitous or a loan or a mixture of them. To be applicable for funding, the project and the organisation must meet certain criteria. Most of the criteria are defined by the Finnish and European Union law, but some criteria are defined by BF themselves. The total amount of the funding has been around 600 million euros annually from which the majority is coming from the government budget of Finland and some part is covered by European Union. At the time of writing, during the COVID-19 crisis in 2020, BF got a mandate to support companies with additional 700 million euros from the supplementary budget.

6.1 Funding process

The process where an applicant is applying funding from BF is called *funding process*. The applicant (customer) can be a business organisation or a research organisation or they can jointly apply funding for joint projects. The applicants should apply funding using an online-service.

The funding process starts when the funding application is submitted using the online-service. The operational phase of the funding process ends when the end report of the project is accepted by BF. Depending of the project, sometimes the customer must also submit reports to BF during the project. Based on the submitted reports and other information BF is verifying that the customer and the project are still qualifying for funding. Sometimes the customer or the project is changing in a way that the criteria for the funding are not met anymore. Because of that BF is cancelling the funding and the customer must return the whole or some part of the already received funding. When the

operational phase of the funding process has ended, BF is still analysing the affects of the funding for the following 5 years and even longer if the project got the support in the form of a loan.

The funding process consists of automatic and manual phases. To be able to serve the customers and the employees of BF more efficiently, BF has build various domain specific systems for automating parts of the funding process. The application development and operations in BF have been outsourced to contractors. However, BF is employing an *enterprise architect* who is making the enterprise level architectural decisions concerning their systems and services.

The heart of the funding process is a massive old monolithic enterprise resource planning system (ERP) implemented in early 2000. The customer relationship management system (CRM) of BF is *tightly coupled* with the ERP. Tightly coupling means that the systems are so dependent on each other that it is really difficult to change one system without affecting the other. Usually when one tightly coupled system is changed the other must be changed as well and they must be tested and installed to production at the same time.

Tight coupling can happen when two or more applications are sharing the same library for communication. Library is an external software component stored in the operating system or bundled together with the application. If the library must be updated in one application and there are incompatible changes in the library, all applications sharing the library must be updated as well. Tight coupling can also happen when the applications are sharing a database. If the structure of the database is changed for one application, it might cause changes to other applications as well.

One way to reduce tight coupling is to employ SOA. In SOA the applications are communicating with each other using service interfaces. Using SOA it is possible to avoid library sharing, however with poor interface design it is still possible to make the applications tightly coupled.

6.2 Service oriented architecture in funding process applications

In 2012 BF started to use enterprise service bus (ESB) for integrating their systems with third party services. The problems caused by tight coupling between the ERP and the CRM made BF to consider using ESB also in the internal service integrations. At that

time their online self-service for the customers was also tightly coupled with their ERP and often it was really hard to change any of these systems without modifications to the other systems as well (Figure 6.1).

All of these three systems, ERP, CRM and online-service, are implemented using a set of standardised specification known as Java enterprise edition (JavaEE) [23]. An application implemented using JavaEE can not be executed alone. The application needs to be installed and executed in a suitable JavaEE *application server* (AS).

Although JavaEE defines the standards for an AS, the implementations of the servers tend to have some implementation specific configurations as well. Platform specific configurations can cause some challenges for changing the implementation of the AS. For example, the JavaEE applications in BF were compiled and packaged using vendor specific tools and even the Java runtime was provided by the vendor. Later on, when BF decided to switch their application server, there were some issues caused by vendor specific tools and configurations.

Enterprise Java Beans (EJB) [10] is component model for implementing JavaEE EJB applications. It defines how two or more JavaEE EJB applications can communicate using *remote method invocation* (RMI) [48]. The communication between JavaEE EJB applications requires that the applications are sharing libraries that are defining the methods and arguments necessary for RMI. The more there are applications sharing the same library, the harder it is to change the methods and arguments defined in the library.

The application containing EJBs must be deployed into AS which supports the specification version of the EJBs in the application. The specification of the EJB has evolved over the time and the vendors of application servers can choose which versions they are supporting. This can create issues when there is a need to change the platform for security, licensing cost or any other reason, since it can require huge changes to the JavaEE applications. In the worst case it is almost impossible to change the application server, which creates a vendor lock-in, and it is also impossible to change the EJB specification, which creates a technology lock-in.

The way how ERP, CRM and online-service JavaEE applications were implemented in BF at 2012 required them all to be on-line at the same time. Because of that, when only one system was updated in production, it was causing downtime to all systems. The installation process at that time was also very slow and error-prone, so updating the production was often causing many hours downtime to all systems.

The problems described in this chapter were among the reasons why BF started to adopt more SOA in their funding process applications. To reduce the complexity and other problems caused by EJB in the applications, BF started to implement RMI using *Spring framework*. Spring framework [24] will be introduced briefly after Figure 6.1. To avoid tight coupling between applications there was now also an option to integrate applications using ESB (Figure 6.2).

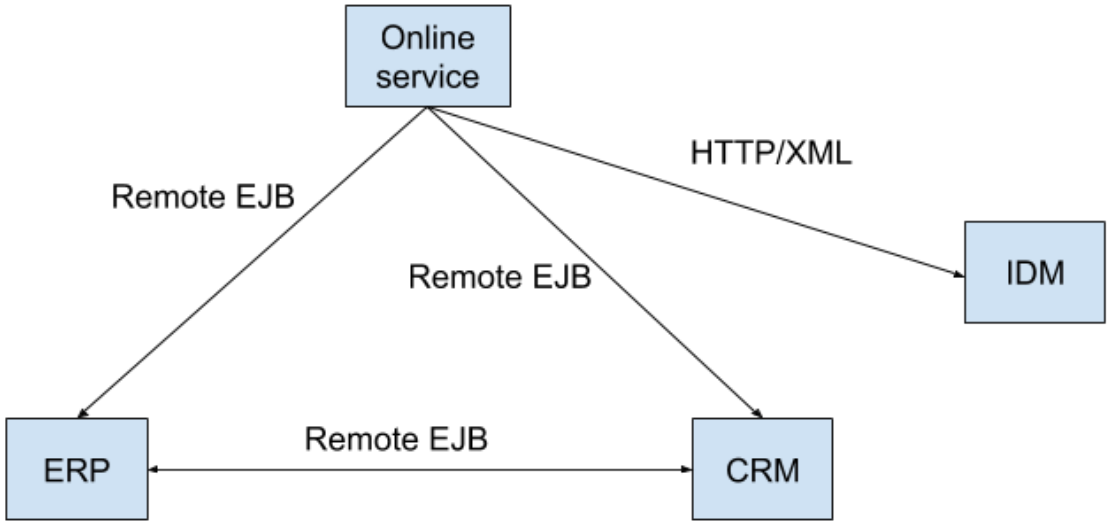


Figure 6.1: BF's architecture before 2012

Spring framework was introduced in 2003 "as a response to the complexity of the early JavaEE specifications" [24]. Using Spring it is possible to use alternative methods for implementing the same functionality in JavaEE application. Spring framework consists of multiple modules and only the necessary modules must be included with the application. It is possible to include Spring modules as libraries in JavaEE application, but using Spring it is also possible to create standalone applications, which do not need to be deployed to any external platform as opposed to JavaEE applications.

Spring HTTP invoker is a lightweight mechanism for implementing RMI. At the abstract level RMI in HTTP invoker and EJB are similar. They both require that the applications are sharing libraries defining the arguments of the interfaces for communication. However, the EJB specification is setting more requirements for the application thus making RMI more complex to implement using EJB. For example, most of the EJB components in BF were implemented using EJB specification version 2.1, which requires a set of different configuration files in addition to the components [9].

To reduce the complexity and tight coupling between the EJB applications, BF decided

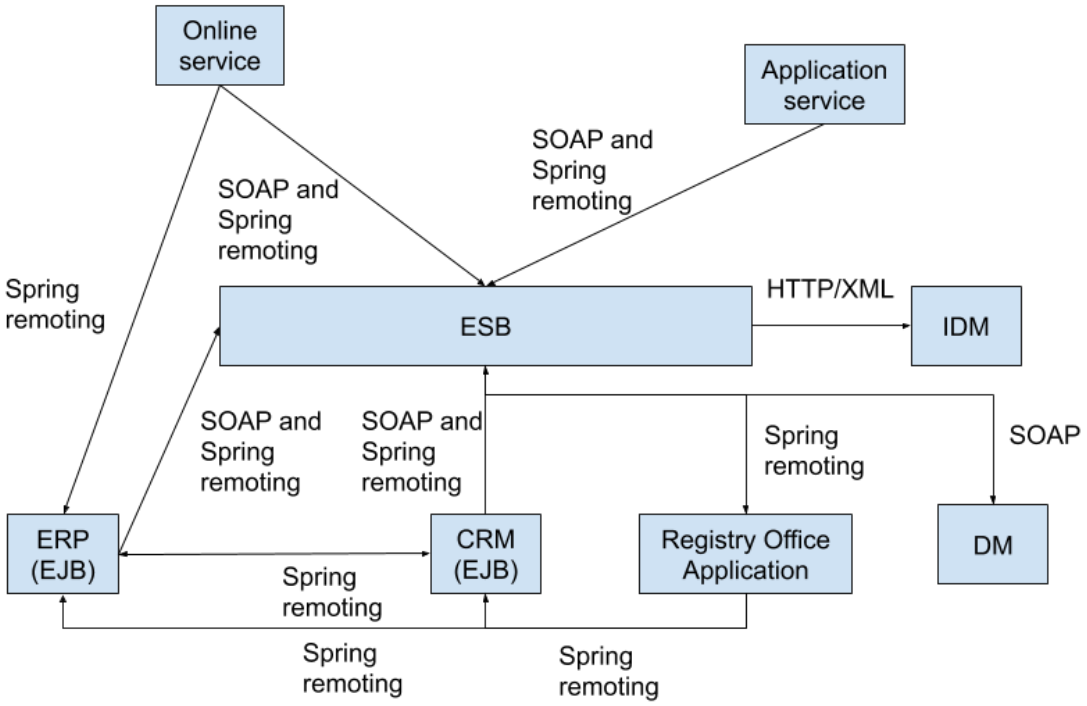


Figure 6.2: SOA in funding process applications

to migrate applications from EJB based RMI to Spring HTTP invoker based RMI. After the migration the applications were using either Spring HTTP invoker or ESB for the communication. This made their ERP, CRM and online-service more independent. That was also a key enabler for JavaEE container implementation change that was made later. Although the migration made it possible to execute the applications on different platforms, there were still shared libraries which were causing some coupling between the applications.

6.3 Microservices in funding process applications

The custom-made CRM in BF is consisting of around 100 000 client records. Although the communication between CRM and ERP was migrated from EJB to Spring HTTP invoker, there is still quite tight coupling between them. The applications are sharing some data in databases and they are also sharing libraries used for communication. The UI of the ERP is even representing some of the data stored in the CRM database and visa versa.

In the organisational structure BF belongs to administrative control of the *Ministry of Economic Affairs and Employment of Finland* also known as TEM. There are also some other organisation belonging to the administrative control of TEM, which are providing

different services to the same clients as BF. Each of these organisations are having their own CRM for storing the customer information. However, since the organisations are sharing the same customers, they are also storing partly the same customer information in their CRM databases. In addition, when the organisations are interacting with the clients, sometimes the organisations need to synchronise the client information with the other organisations. For these reason, TEM has started a project where the organisations under their administration should start to use a shared database for exchanging the client information. In addition, TEM is providing a proprietary CRM product for the organisations under its control for managing the customer data.

Although the organisations under the administrative control of TEM can access the information stored in the shared database and the organisations can use the same CRM product, BF can not discard their old custom-made CRM. The proprietary CRM will not be able to provide all the required services for the ERP and other systems in BF. However, the majority of BF custom-made CRM functions are covered by the proprietary CRM. This was one of the reasons why BF decided to stop developing and maintaining their custom-made CRM. The lack of required services in the proprietary CRM and the decision to stop developing and maintaining their custom-made CRM, created a need to implement soon disappearing services using other methods. The positive experiences with SOA and Spring framework and also the rise of MSA style in general were leading BF to implement required services in MSA style.

One of he first microservice implementations was an application for managing customers bank account numbers. Although customer bank account was stored in the CRM, the problem was that BF requires that the clients can administrate their bank accounts by themselves. Their custom-made CRM did not have this kind of capability and it was unclear when the proprietary CRM will have it or will it have it at all. This led to a microservice Proof Of Concept (POC) project. The aim of the project was to migrate the customer bank account number management to a new 'Bank account number management' named microservice. There was also a requirement to remove related functionality from their custom-made CRM and to integrate the new microservice with their ERP in a similar way like it was integrated with the custom-made CRM.

In MSA an UI is usually a compositional layer for multiple microservices and the UI application should be deployed separately. However, in the POC project in BF the UI application was bundled to the same deployment unit with the actual microservice (Figure 6.3). The UI application is implemented using Javascript and React framework and it

has Spring boot framework as a backend. The communication between the frontend and the backend is implemented using *REpresentational State Transfer* (REST) pattern. The actual microservice is a separate software module also implemented using Spring Boot framework. It could be deployed as an individual micorservice application, but now it is bundled together with UI application backend. Bundling the UI application together with the actual microservice seemed harmless at the beginning. However, the bundling is against MSA ideology. Consequently the bundiling has caused already some problems which could have been avoided using MSA patterns (see Figure 6.4 or Figure 6.5).

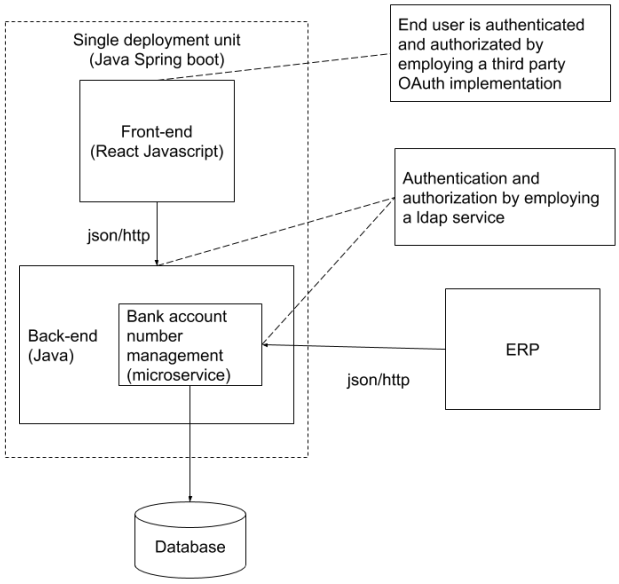


Figure 6.3: BF's Bank account number management application.

One of the problems caused by bundling the UI application together with the microservice is the complexity related to security configuration of the system. Although the UI application and microservice code are in separate modules, the application is executed as a single unit. The UI application and the microservice module have different requirements for authentication and authorisation. The end user of the UI application is authenticated and authorised utilizing third party service with OAuth protocol. Other systems calling the microservice are authenticated and authorised utilising *Lightweight Directory Access Protocol* (LDAP) server.

OAuth and LDAP authentication and authorisation are implemented in different libraries in Spring framework. They require certain versions of Spring framework core security libraries. In some situations upgrading OAuth or LDAP library may require upgrading

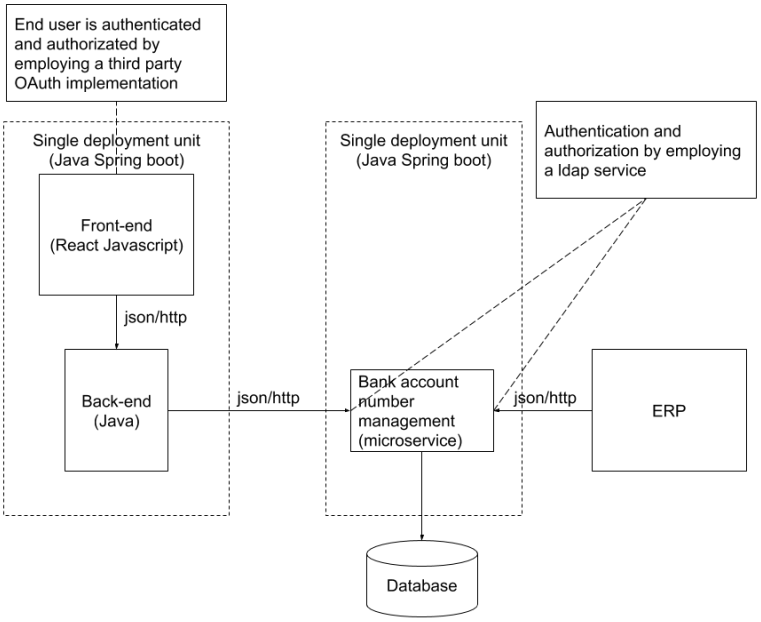


Figure 6.4: Alternative reference implementation using MSA.

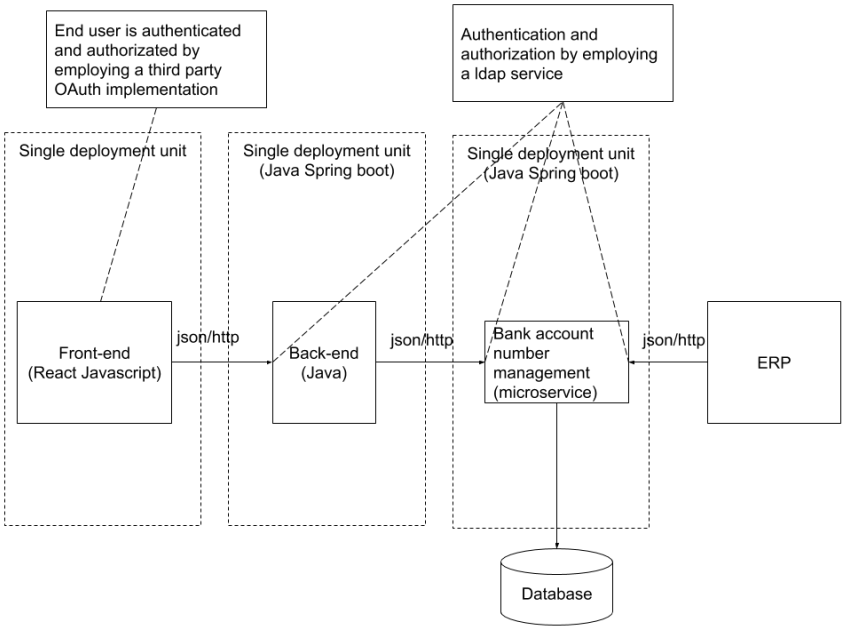


Figure 6.5: Alternative reference implementation using MSA.

the other as well if the shared core libraries would be different. It is also possible that there is no suitable library published for the other library and thus the libraries can not be upgraded at all. This could cause even a security threat.

The code that does the authentication and authorisation became unnecessarily complicated and expensive to maintain. If the user interface was a separate deployment unit the

security configuration would be simpler and it would reduce the maintenance costs [45].

When an organisation starts to adopt microservices, the organisation needs to make some investment for the runtime environment and deployment pipeline. Microservices are consuming more system resources and the amount of deployment units is drastically increased compared to monolithic applications. When the first microservices were introduced in BF, the DevOps processes were still under development.

Although microservice are cloud-native and relatively easy to deploy into multiple servers, BF had only one virtual server with fixed system resources for the applications in production. There has already been problems with the memory and the disk consumption earlier. The lack of system resources and DevOps processes led to the decision to try to minimise the amount of the deployment units in this particular case.

When BF started their migration from their custom-made CRM to the shared CRM, it become clear that there was a need for some kind of database and service for some data that can not be immediately migrated to the shared CRM. There was also worry about the reliability of the integration with the shared CRM. This led to the decision to implement 'Organization Service' microservice which would be responsible for storing the data that can not be immediately migrated to the shared CRM.

The worry about system resource consumption and because the deployment pipeline was still under development, BF made a decision to implement 'Organization Service' in the same deployment unit with 'Bank account number management' application. The decision was also supported by the fact that both of these microservices could share the same database schema.

Database sharing should not be used in microservices [43]. To avoid tight coupling via database each microservice should have its own database. When BF started to implement microservices they had only a few databases in use and those were shared in legacy applications. At that time, the management of the databases was outsourced to a third party and the process for creating a new database was slow and error prone. It could take weeks or even months from the order before a new database was available. If there were some problems in the database creation order or in the properties of the created database, the whole process must be started again. The difficult process for ordering new databases and the need for rapid implementation of microservices led to a decision to share databases between microservices.

6.4 DevOps evolution in Business Finland

According to Google Trends the interest towards MSA and DevOps has been growing at equal rate since 2014 [3]. At that time, the trend was also followed by BF as they started to evaluate a tool for operating system provisioning and application deployment. Earlier the application deployment in BF was quite complicated manual operation. At first the application was compiled and tested in Jenkins CI and then if the build was successful the artifact was deployed to Maven repository. After that the artifact was manually transferred to the test environment. Application configuration files were modified manually, old application was stopped, database was updated and then new application was started. Then the application was tested by BF and if it passed testing the same complicated deployment process was done in production server. The process was complicated, slow and error prone and that was one of the reasons why BF wanted the amount of applications to be low.

At the same time, when BF started to evaluate a tool for the application deployment, there was also a new requirement for software functionality as described in chapter 6.3. However, the DevOps practices in BF were not yet ready for managing more than a few applications. In addition, the organisation structure and the software development culture in BF was not ideal for MSA development. There were two large development teams in different locations maintaining half a dozen monolithic applications. The software development tasks were distributed evenly between the teams and all developers were able to make changes to any application.

After adopting more DevOps practises and changing the SOA development style more towards MSA style, the organisation structure in BF software development has changed. Instead of having only two teams there are now multiple miniteams. When a new service is implemented, a new team is organised for developing the new service. The team is owning the service as long as it is actively developed. This means that the team is implementing requested changes to the service. However, after the service has been installed into production and all the major issues related to production are resolved, the maintenance team takes the responsibility. This does not follow the "you build it, you run it" philosophy which is encouraged in MSA.

Software development in BF is implemented following continuous delivery pattern. Each change in VCS is triggering a new build in CI. The build is executing automated unit and integration tests. If the software is passing the tests, CI is installing the software into CI

server. After that, CI is testing the software by executing automated E2E and UI tests. If any of the build stage is failing, the developers can see it immediately from the UI of the CI server or after a short delay from an external monitoring service.

The installation of the software into different environments is made in the same way by employing DevOps practices. If the installation succeeds in CI, it gives a great confidence that the installation to test and production environments will succeed as well. Installation to test environments is done automatically after a file containing the version number of the software is updated in VCS. Installation into production environment is done manually executing the same commands as in automatic installation into test environment.

7 Conclusions

Microservice architectural style (MSA) was introduced in 2011 and today it has become one of the most popular application development styles. However, MSA is sharing so many patterns with *service oriented architecture* (SOA) that is not describing an entirely new architectural style. It should rather be seen as a specific way to design and implement SOA. For example, before the term microservice was introduced, Netflix as one of the early adaptors of microservices, was describing their microservice based systems as *fine grained SOA*. In addition some others were calling microservice style as "SOA done right".

Some organisations are seeing microservices to provide solutions to some of the problems with their existing monolithic applications. The organisations are planning to migrate their monolithic applications into microservices, but it is not clear when such a migration is advantageous. Switching a monolithic application to microservices is expensive and it increases the complexity of the whole system. However, since microservices are having so many beneficial aspects, in a long run, for some organisations the migration can be cost-efficient.

Compared to monolithic application, microservices are easier to maintain, since the functionality of a single microservice is limited and easier to understand. If there is an issue in the system, it is easier isolate the problematic functionality in the codebase of a single microservice than in the codebase of a large monolithic system. In addition it is easier and faster to resolve the issue, since a single microservice is independent thus it is possible to test and deploy only the problematic part of the system.

Although a problem in a MSA application can be easier to resolve than in a monolithic application, finding the cause of the problem is much more difficult. Since MSA application consists of multiple services, which can be distributed, it can be more challening to find the problematic service compared to a monolithic application. Also the distributed nature of a MSA application makes the whole system more complex than a monolithic application.

The complexity related to *enterprise service bus* (ESB) can create both a technology and a vendor lock-in. For example, in Business Finland (BF) the services implemented in the ESB are created using vendor specific configuration. For that reason, it is challenging to change the vendor of the ESB. In addition, because of the complexity related to the service implementation and the environment, the developers in BF are avoiding to change

them in any way. Updating the ESB or even any library for some service is complicated. It is not even possible to update the runtime environment for the ESB. As a result of this the whole system can get outdated and it can cause even a security risk.

In contrast to SOA, MSA is discouraging the use of ESB. In MSA the services are kept cohesive and common business logic implemented in ESB is not supporting that idea. Since BF is using ESB only for routing some service calls, it would be quite easy to remove. After the removal of the ESB, the architecture of BF would be even closer to MSA style SOA.

Bundling microservices to the same deployment unit in BF has resulted to a monolithic modular application instead of independent microservices. Those modules could be extracted again to independent microservices to resolve some of the problems caused by monolithic application.

The decision to use HTTP and JSON for integrating applications in BF has decreased the coupling between the applications. It allows application development without sharing any libraries thus reducing dependencies and making the application more independent. It also enables polyglot programmin in the application development. Consequently, the teams can develop the applications more efficiently since the teams can choose the best suitable technologies. The increased independency in the services enables teams to develop and deploy services without dependencies. This allows BF to deploy new functionality and improvements to their software in a lean way.

Automatisation of *development and operations* (DevOps) is essential in microservice style. With automatisation it is possible to reduce the complexity related to the distributed nature of microservice style applications. Automatisation enables *continuous delivery*, so new features and improvements to the applications can be delivered to the end users in lean and rapid fashion.

The automatisation of the deployment pipeline in BF has enabled teams to rapidly publish new versions of applications. The applications are always in a stage that a new release and production installation can be made in matter of minutes. Using the same binaries and installation automation to all environment gives a great confidence that the production installation will succeed and, indeed, the configuration errors during the installations have decreased drastically. Automatised deployment pipeline has enabled *continuous delivery* and has improved the quality, since the applications are automatically tested after each change it the source code. Automated testing provides fast feedback about possible new problems thus reducing the cost for fixing the problem.

The current architecture in BF is maybe closer to traditional SOA than MSA. However, the problems caused by the ESB and the tight coupling of the services have been influencing the transition towards MSA. This supports the idea that MSA is ”SOA done right” in this particular case.

References

[1] N. Alshuqayran, N. Ali, and R. Evans. "A Systematic Mapping Study in Microservice Architecture". In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. 2016, pp. 44–51.

[2] *Ansible, Tool for software provisioning.* Accessed on 16.8.2020. URL: https://www.ansible.com/.

[3] A. Balalaie, A. Heydarnoori, and P. Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52.

[4] J. Bogner et al. "Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019, pp. 187–195.

[5] J. Bogner, A. Zimmermann, and S. Wagner. "Analyzing the Relevance of SOA Patterns for Microservice-Based Systems". In: *In Proceedings of the 10th Central European Workshop on Services and their Composition (ZEUS'18)*. CEUR-WS, Apr. 2018, pp. 9–16.

[6] *Business Finland.* Accessed on 22.10.2020. URL: https://www.businessfinland.fi/.

[7] *Chef, Tool for software provisioning.* Accessed on 16.8.2020. URL: https://www.chef.io/.

[8] M. Cohn. *Succeeding with Agile: Software Development Using Scrum.* Second printing. Pearson Education, Inc, Jan. 2010.

[9] L. G. DeMichiel. *Enterprise JavaBeans Specification, Version 2.1.* Sun Microsystems. Oct. 2003.

[10] L. DeMichiel and M. Keith. *Enterprise JavaBeans Specification, Version 3.0.* Sun Microsystems. May 2006.

[11] A. Deutschman. *Inside the Mind of Jeff Bezos.* Accessed on 15.9.2020. Aug. 2004. URL: http://www.fastcompany.com/magazine/85/bezos_1.html.

[12] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley, 1999.

[13] *Finpro and Tekes uniting as Business Finland*. Accessed on 22.10.2020. URL: https://www.businessfinland.fi/en/whats-new/news/2017/Finpro-and-Tekes-uniting-as-Business-Finland/.

[14] M. Fowler. *Continuous Integration*. Accessed on 19.4.2020. 2006. URL: https://www.martinfowler.com/articles/continuousIntegration.html.

[15] M. Fowler. *ContinuousDelivery*. Accessed on 10.9.2020. 2013. URL: https://www.martinfowler.com/bliki/ContinuousDelivery.html.

[16] M. Fowler. *Microservices*. Accessed on 19.4.2020. 2014. URL: https://martinfowler.com/articles/microservices.html.

[17] M. Fowler. *Xunit*. Accessed on 31.8.2020. 2006. URL: https://martinfowler.com/bliki/Xunit.html.

[18] *Git, Verion Control System*. Accessed on 11.11.2020. URL: https://git-scm.com/.

[19] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, Inc, 2010.

[20] *IBM Rational ClearCase, Verion Control System*. Accessed on 11.11.2020. URL: https://www.ibm.com/products/rational-clearcase.

[21] R. Jabbari et al. "What is DevOps? A Systematic Mapping Study on Definitions and Practices". In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP '16 Workshops. Edinburgh, Scotland, UK: Association for Computing Machinery, 2016. ISBN: 9781450341349. DOI: 10.1145/2962695.2962707. URL: https://doi.org/10.1145/2962695.2962707.

[22] D. Janzen and H. Saiedian. "Test-driven development concepts, taxonomy, and future direction". In: *Computer* 38.9 (2005), pp. 43–50.

[23] *Java EE Platform specification*. Accessed on 24.10.2020. URL: https://javaee.github.io/javaee-spec/Specifications.

[24] R. Johnson et al. *Spring Framework Documentation*. Accessed on 25.10.2020. URL: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/index.html.

[25] *JUnit, Testing framework*. Accessed on 31.8.2020. URL: https://junit.org.

[26] K. Karhu et al. “Empirical Observations on Software Testing Automation”. In: *2009 International Conference on Software Testing Verification and Validation*. 2009, pp. 201–209.

[27] M. C. MacKenzie et al. “Reference Model for Service Oriented Architecture 1.0”. In: *OASIS Standard* (Oct. 2006).

[28] P. Mell and T. Grance. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology. Sept. 2011.

[29] *Mercurial, Verion Control System*. Accessed on 11.11.2020. URL: https://www.mercurial-scm.org/.

[30] G. Meszaros. *xUnit Test Patterns Refactoring Test Code*. Pearson Education, Inc, 2007.

[31] N. Dragoni and others. *Microservices: Yesterday, Today, and Tomorrow*. Ed. by M. Mazzara and B. Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12.

[32] N. Kratzke and P-C. Quint. “Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study”. In: *Journal of Systems and Software* 126 (2017), pp. 1–16. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.01.001. URL: http://www.sciencedirect.com/science/article/pii/S0164121217300018.

[33] S. Neely and S. Stolt. “Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)”. In: *2013 Agile Conference*. 2013, pp. 121–128.

[34] S. Newman. *Building Microservices*. 1st. O’Reilly Media, Inc., Feb. 2015.

[35] C. O’Hanlon. “A Conversation with Werner Vogels: Learning from the Amazon Technology Platform: Many Think of Amazon as ’That Hugely Successful Online Bookstore.’ You Would Expect Amazon CTO Werner Vogels to Embrace This Distinction, but in Fact It Causes Him Some Concern.” In: *Queue* 4.4 (May 2006), pp. 14–22. ISSN: 1542-7730. DOI: 10.1145/1142055.1142065. URL: https://doi.org/10.1145/1142055.1142065.

[36] C. Pahl and P. Jamshidi. “In Proceedings of the 6th International Conference on Cloud Computing and Services Science”. In: *Microservices: A Systematic Mapping Study*. SCITEPRESS, 2016, pp. 137–146.

[37] *Puppet, Tool for software provisioning*. Accessed on 16.8.2020. URL: https://puppet.com/.

[38] *Robot Framework, User Interface test tool*. Accessed on 12.11.2020. URL: https://robotframework.org/.

[39] P. Rodríguez et al. "Continuous deployment of software intensive products and services: A systematic mapping study". In: *Journal of Systems and Software* 123 (2017), pp. 263–291. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2015.12.015. URL: http://www.sciencedirect.com/science/article/pii/S0164121215002812.

[40] P. Runeson. "A survey of unit testing practices". In: *IEEE Software* 23.4 (2006), pp. 22–29.

[41] *Selenium, User Interface test tool*. Accessed on 12.11.2020. URL: https://www.selenium.dev/.

[42] *Silk Test, User Interface test tool*. Accessed on 12.11.2020. URL: https://www.microfocus.com/en-us/products/silk-test/overview.

[43] D. Taibi and V. Lenarduzzi. "On the Definition of Microservice Bad Smells". In: *IEEE Software* 35.3 (2018), pp. 56–62.

[44] D. Taibi, V. Lenarduzzi, and C. Pahl. "In Proceedings of the 8th International Conference on Cloud Computing and Services Science". In: *Architectural Patterns for Microservices: A Systematic Mapping Study*. SCITEPRESS, 2018, pp. 221–232.

[45] D. Taibi, V. Lenarduzzi, and C. Pahl. "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation". In: *IEEE Cloud Computing* (Sept. 2017), pp. 22–32.

[46] D. Wampler and T. Clark. "Guest Editors' Introduction: Multiparadigm Programming". In: *IEEE Software* 27.5 (2010), pp. 20–24. DOI: 10.1109/MS.2010.119.

[47] J. A. Whittaker. "What is software testing? And why is it so hard?" In: *IEEE Software* 17.1 (2000), pp. 70–79.

[48] A. Wollrath, R. Riggs, and J. Waldo. "A Distributed Object Model for the Java System". In: *Proceedings of the USENIX 1996*. 1996.

[49] O. Zimmermann. "Microservices tenets". In: *Computer Science - Research and Development* 32.Issue 3-4 (July 2017), pp. 301–310. DOI: https://doi.org/10.1007/s00450-016-0337-0.