Eberhard Wolff

Microservices Primer

A Short Overview

Microservices Primer

A Short Overview

Eberhard Wolff

This book is for sale at http://leanpub.com/microservices-primer

This version was published on 2018-04-25

Tweet This Book!

Please help Eberhard Wolff by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought the Microservices Primer by @ewolff

The suggested hashtag for this book is #MicroservicesPrimer.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#MicroservicesPrimer

Also By Eberhard Wolff

Microservices - Ein Überblick

Microservices Rezepte

Microservices - A Practical Guide

Microservices Recipes

Contents

1 Introduction

Microservices have turned into a hype. Like for every other architecture approach there are certain scenarios where the advantages of Microservices can best unfold. Besides, their characteristic advantages come at the price of challenges in other areas.

This primer provides a short introduction into the topic of Microservices. It presents what Microservices actually are and which advantages they can have. This overview is meant to facilitate the start into the topic and to help with judging the applicability and the usefulness of Microservices in a certain context. Thereby Microservices can turn from just a hype into a meaningful element in an architect’s tool kit.

2 What Are Microservices?

The idea behind Microservices is not new. A very similar approach is also followed by the UNIX philosophy, which is based on three ideas:

- A program should fulfill only one task, and it should do it well.
- Programs should be able to work together.
- Besides, the programs should use a universal interface. In UNIX these are text streams.

The realization of these ideas leads to the creation of reusable programs, which are in the end a kind of component.

Microservices serve to divide large systems. Consequently, Microservices represent a modularization concept. There is a large number of such concepts, but Microservices are different. They can be brought into production independently of each other. Changes to an individual Microservice only require that this Microservices has to be brought into production. In the case of other modularization concepts all modules have to be delivered together. Thus, a modification to an individual module necessitates that the entire application with all its modules has to be deployed again.

Microservice = Virtual Machine

Microservices cannot be implemented via the modularization concepts of programming languages. These concepts usually require that all modules have to be delivered together in one program. Instead Microservices have to be implemented as virtual machines, as more light-weight alternatives such as Docker containers or as individual processes. Thereby they can all easily be brought into production individually.

This results in a number of additional advantages: For instance, Microservices are not bound to a certain technology. They can be implemented in any programming language or on any platform. Moreover, Microservices can of course also bring along their own supporting services such as databases or other infrastructure.

Besides, Microservices should possess their own separate data storage i.e. a separate database or at least a separate schema in a common database. Consequently, each Microservice is in charge of its own data. In fact, experience teaches that the shared use of database schemas renders changes to the data structures practically impossible. Since this interferes profoundly with software changeability, this kind of coupling should be prevented.

Communication Between Microservices

Microservices have to be able to communicate with each other. This can be achieved in different manners:

- The Microservices can replicate data. This does not just mean to copy the data without changing the schema. In that case changes to the scheme are impossible because multiple Microservices use the same schema. However, when one Microservice processes orders and another analyzes the data of the orders, the data formats can be different and also access to the data is different: The analysis Microservice will primarily read data, for order processing reading and writing are rather equal. Classical data warehouses also employ replication for analyzing large amounts of data.
- When Microservices possess an HTML UI, they can easily use links to other Microservices. Besides, it is possible that a Microservice integrates the HTML code of other Microservices in its own web page.

- Finally, the Microservices can communicate with each other by protocols like REST or messaging via the network.

In a Microservice-based system it has to be defined which communication variants are used to ensure that the Microservices can in fact be reached with these technologies.

2.1 Size

The term “Microservice” focuses on the size of Microservices. This makes sense for distinguishing Microservices from other definitions of “services”. Nevertheless, it is not so easy to indicate the concrete size of Microservices.

Defining the unit poses already a problem: Lines of Code (LoC) are not a good unit. In the end, the actual number of Lines of Code of a program does not only depend on its formatting, but also on the programming language. In fact, it does not seem to make much sense to evaluate an architectural approach based on such metrics. Ultimately, the size of a system can hardly be given in absolute terms, but only in relation to the represented business processes and their complexity.

Therefore, it makes much more sense to define the size of Microservices with the aid of upper and lower limits. In general, it holds true that smaller is better for Microservices:

- A Microservice should be developed by one team. Therefore, a Microservice should never be so large that more than one team is necessary to develop it further.
- Microservices represent a modularization approach. Developers should be able to understand individual modules – therefore modules and thus Microservices have to be so small that an individual developer is still able to comprehend them.

- Finally, a Microservice should be replaceable. When a Microservice cannot be maintained anymore or for instance a more powerful technology is supposed to be used, the Microservice can be replaced by a new implementation. Microservices represent therefore the only software architecture approach which takes a future replacement of the system or at least of system parts already into consideration during development.

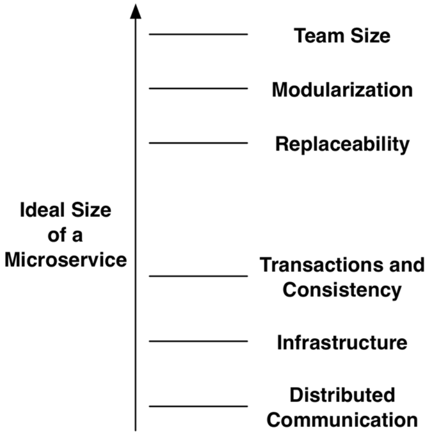
Team Size
Modularization
Replaceability
Ideal Size of a Microservice
Transactions and Consistency
Infrastructure
Distributed Communication

Fig. 1: Ideal Size of a Microservice

This leaves the question why not to just build the Microservices as small as possible. In the end, the advantages reinforce each other when the Microservices are especially small. However, there are different reasons why Microservices cannot be tiny without creating also a number of problems:

- Distributed communication between Microservices via the network is expensive. When Microservices are large, the communication occurs rather locally within a Microservice and is therefore faster and more reliable.

- It is difficult to move code across Microservice boundaries. The code has to be transferred into another system. When this system uses a different technology or programming language, rewriting the code in a different language might be the only option for moving a functionality from one Microservice into another. Of course, it is always possible to turn the respective functionalities into a new Microservice, which can be accessed by the other Microservices. In contrast, within a Microservice refactoring is quite easy with the aid of the usual mechanisms e.g. automated refactoring in the IDE.
- A transaction within a Microservice is easy to implement. Beyond the boundaries of an individual Microservice this is not trivial anymore since distributed transactions become necessary. Therefore the best is to decide for a Microservice size which allows that a transaction can be entirely processed in one Microservice.
- The same holds true for the consistency of data: When for instance the account balance is supposed to be consistent with the result of earnings and expenses, this can quite easily be implemented in one Microservice, but is hardly feasible across Microservices. Therefore, Microservices should be large enough to ensure that data which have to be consistent are handled in the same Microservice.
- Each Microservice has to be brought into production independently and therefore needs its own environment. This uses up hardware resources and means in addition that the effort for system administration increases. When there are larger and therefore fewer Microservices, this expenditure becomes smaller.

To a certain degree the size of a Microservice depends on the infrastructure: When the infrastructure is very simple, it can support a multitude of Microservices and therefore also very small Microservices are possible. In such a case the advantages of a

Microservice-based architecture are accordingly larger. Already relatively simple measures can help to reduce the infrastructure expenditure: When there are templates for Microservices or other possibilities to create Microservices easily and to more uniformly administrate them, this can already reduce expenditure and thereby enable the use of smaller Microservices.

Nanoservices

Certain technological approaches can further reduce the size of a service. Instead of delivering Microservice as virtual machines or Docker containers, the services can be deployed on Amazon Lambda[1]. It allows the deployment of individual functions written in Java, Node.js or Python. Each function is automatically monitored. In addition, each call to a function is billed. Functions can be called using REST or due to events e.g. data written to Amazon S3 or DynamoDB. Using such an infrastructure makes it possible to create services that just consist of a few Lines of Code each because the overhead for deployment and operations is so low.

A similar approach can be implemented using a Java EE application server. Java EE defines different deployment formats and allows to run multiple applications on an application server. The services communicate for instance via REST or messaging just like Microservices. However, in such a scenario the services are not so well isolated from each other anymore: When an application in an application server uses up a lot of memory, this will also affect the other applications on the application server.

Another alternative are OSGi bundles. This approach also defines a module system based on Java. However, in contrast to Java EE this approach allows method calls between bundles so that communication via REST or messaging is not necessarily required.

Unfortunately, both approaches are problematic when it comes to

---

[1] https://aws.amazon.com/lambda/

independent deployment: In practice, Java EE application servers and also OSGi runtime environments have often to be started again when new modules are deployed. Therefore, a deployment affects also other modules.

On the other hand, the expenditure for infrastructure and communication is lower since OSGi allows for instance to use local method calls. This enables the use of smaller services.

To clearly distinguish these services from Microservices it is sensible to use an alternative term like "Nanoservices" for this approach. Ultimately these services offer neither the isolation of Microservices nor their independent deployment.

2.2 Bounded Context and Domain-Driven Design

It is one of the main objectives of Microservices to limit changes and new features to one Microservice. Such changes can comprise the UI – therefore a Microservice should also provide a UI. However, also in another area modifications should occur within the same Microservice – namely in regards to data.

A service which implements an order process should ideally also be able to query and modify the data for an order. Microservices have their own data storage and can therefore store data in the way that best suits them. However, an order process requires more than just the data of the order. Also the data concerning the customer or the items are relevant for the order process.

Here, Domain-driven Design ((DDD)[2]) is helpful. Domain-driven Design serves to analyze a domain. The essential basis is *Ubiquitous Language*. This is like other components of Domain-driven Design

---

[2] Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley,2003, ISBN 978-0-32112-521-7

also a pattern and therefore here set in *italics*. *Ubiquitous Language* denotes the concept that everybody who participates in the software should use the same terms. Technical terms like order, bill etc. should be directly echoed in the software. Often enterprises have their own specific language – this language should then also be implemented in the software.

The domain model can consist of different elements:

- *Entity* is an object with its own identity. In an E-commerce application the customer or the item could be *Entities*. *Entities* are typically stored in a database.
- *Value Objects* do not have their own identity. An example is an address which only makes sense in the context of a specific customer and therefore does not possess an identity.
- *Aggregates* are composite domain objects. They enable a simpler handling of invariants and other conditions. For instance, an order can be an *Aggregate* of order lines. This allows for instance to ensure that the order of a new customer does not surpass a certain limit. The invariant has to be fulfilled via a calculation of values from the order lines so that the order as *Aggregate* can ensure these conditions.
- *Services* contain business logic. DDD focuses on the modeling of business logic as *Entities*, *Value Objects* and *Aggregates*. However, logic which accesses multiple of these objects cannot be modelled in one of these objects. For this purpose there are *Services*.
- *Repositories* serve to access the entirety of all *Entities* of a type. Typically, some kind of persistence, for example in a database, is what is used to implement a *Repository*.

The implementation of a domain model from these components and also the idea of *Ubiquitous Language* facilitate the design and development of object-oriented systems. However, it is not immediately clear which relevance DDD might have for Microservices.

Bounded Context

Domain-driven Design does not only provide a guideline for how a domain model can be implemented, but also for the relationships between domain models. Having multiple domain models initially appears unusual. After all, concepts like customer and order are central for the entire enterprise. Therefore, it seems attractive to implement exactly one domain model and to carefully consider all aspects of the model. This should make it easy to implement the software systems in the enterprise based on these elements.

However, *Bounded Context* states that such a general model cannot be implemented. Let's take the customer of the E-commerce shop as an example: The delivery address of this customer is relevant in the context of the delivery process. During the order process, on the other hand, the specific preferences of the customer matter, and for billing the options for paying, for which the customer has deposited data, are most important – for example his/her credit card number or information for a direct debit.

Theoretically it might be possible to collect all this information in a general customer profile. However, this profile would then be extremely complex. Besides, in practice it would not be possible to handle it: When the data regarding one context change, the model needs to be changed and this will then concern all components which use the customer data model – and these can be numerous. In addition, the analysis needed to arrive at such a customer model would be so complex that it would be hard to achieve in practice.

Bounded Context and Microservices

Therefore a domain model is only sensible in a certain context – i.e. in a *Bounded Context.* For Microservices it makes the most sense to design them in a way that each Microservice corresponds to a *Bounded Context.* This provides an orientation for the domain architecture of Microservices. This design is especially important since a good domain architecture enables independent work on features. When the domain architecture ensures that each feature is implemented in an individual Microservice, the implementation of features can be uncoupled. Since Microservices can even be brought into production independently of each other, features cannot only be developed separately, but also rolled out individually.

The independent development of features also profits from the distribution in *Bounded Contexts*: When a Microservice is also in charge of a certain part of the data, the Microservice can introduce features without causing changes to other Microservices.

When for example in an E-commerce system a payment option via PayPal is supposed to be introduced, this requires only changes to the Microservice for billing thanks to *Bounded Context*. There the UI elements and the new logic are implemented. As the Microservice for billing administrates the data for the *Bounded Context*, only the PayPal data have to be added to the data of the Microservice. Changes to a separate Microservice, which administrates the data, are not necessary. Therefore, the *Bounded Context* is also an advantage in regards to changeability.

Relationships Between Bounded Contexts

In his book Eric Evans describes different manners how *Bounded Contexts* can work together. In the case of *Shared Kernel* for instance a shared *Bounded Context* can be used in which the shared data are stored. A radical alternative option is *Separated Ways*: Here, both *Bounded Contexts* use completely independent models. *Anticorruption Layer* uncouples two domain models. Thereby it is for instance possible to prevent that an old and hard to understand data model from a mainframe has to be used in the remainder of the system. With the aid of an *Anticorruption Layer* the data are transferred into a new, easily understandable representation.

Of course, depending on the model used for the relationships between the *Bounded Contexts* more or less communication between the teams working on the Microservices will be necessary.

In general, it is therefore conceivable that a domain model also comprises multiple Microservices. Maybe it is sensible in the example of the E-commerce system that the modeling of the basic data of a customer is implemented in one Microservice and only specific data are stored in the other Microservices – following the concept of *Shared Kernel*. However, in such a case more coordination between Microservices might be necessary which can interfere with their separate development.

2.3 Conway's Law

Conway's Law[3] was coined by the American computer scientist Melvin Edward Conway and states:

> Organizations which design systems can only create such designs which reflect the communication structures of these organizations.

The reason behind Conway's Law is that each organizational unit designs a certain part of the architecture. If two parts of the architecture are supposed to have an interface, coordination is required regarding this interface – and therefore a communication relationship between the organizational units which are responsible for the respective parts.

The Law as a Limit for Architecture

An example for the effects of the Law: An organization forms one team of experts each for the web UI, the logic in the backend and

---

[3] http://www.melconway.com/research/committees.html

for the database (compare Fig. 2). This organization has advantages: The technical exchange between experts is relatively simple and holiday replacements are easy to organize. Therefore, the idea to have employees with similar qualification work together in one team is not very far fetched.

However, according to Conway's Law the three teams will create artifacts in the architecture. A database layer, a backend layer and a UI layer will be created.

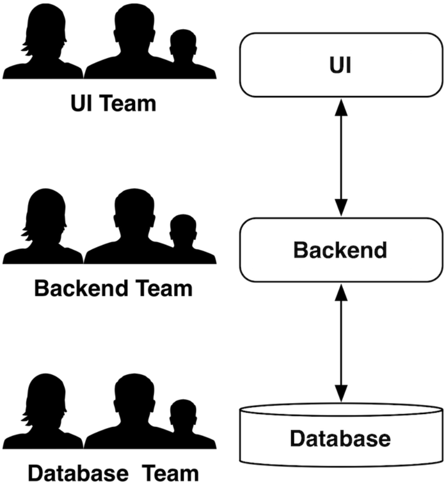

Fig. 2: Team setup by technical skills

This architecture entails a number of disadvantages:

- To implement a feature, the customer has to talk with all three teams. He/she has to explain the requirements for the new feature to each of the three teams. When the customer does not have detailed knowledge about the system architecture, the teams will have to discuss with him/her how the functionalities can be implemented in the different layers.
- The teams have to coordinate their work for instance in regards to interfaces.

- In addition, it has to be ensured that each team delivers its part of the work in time. The backend can hardly implement new features without changes to the database. Likewise, the UI cannot be implemented without changes to the backend.
- The dependencies and the resulting need for communication slow down the implementation of the feature. The database team can only deliver the changes at the end of its sprint. As the work of the backend team is based on the changes introduced by the database team, it has to wait until the other team is done. Likewise, the UI team has to wait for the backend team to finish. Therefore, it might take three sprints until the entire implementation is completed. Of course, optimizations are possible. However, a completely parallel implementation is practically impossible.

The way the teams were created results therefore in an architecture which interferes with a faster implementation of features. This problem is often not perceived, as the relationship between organization and architecture is frequently not considered.

Conway's Law as Enabler

However, it is also possible to deal very differently with Conway's Law. It is the explicit aim of Microservices to implement a domain architecture where each Microservice implements a meaningful part of the architecture e.g. a *Bounded Context*. That facilitates and parallelizes the work on domain aspects. Therefore, there is another way to handle Conway's Law in the context of Microservices: Instead of letting the architecture be a result of the organization, the architecture drives the organization. The organization gets structured in the way that best supports the architecture.

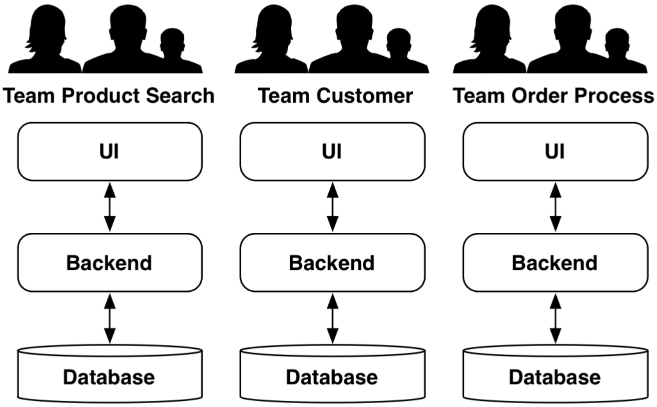

Fig. 3: Team setup by functionalities

Fig. 3 shows a possible architecture for a Microservice-based system: There is one component each for product search, the handling of customers and the order process. For each Microservice there is a team which implements this Microservice. Thereby the domain-based distribution into Microservices is not only implemented at the level of architecture, but also at the organizational level. This supports the architecture: Transgressing the architecture gets difficult because according to Conway's Law the organization enforces a domain-based architecture.

Nevertheless, the different technical artifacts have to be implemented in the Microservices. Accordingly, the required technical skills have to be present in the different teams. Within a team the coordination of experts is profoundly easier than across teams. For this reason, requirements which have to be implemented across different technical artifacts are now easier to implement.

Organizational Compromises

In practice, even in a system structured as described with a supporting organization there are nevertheless still challenges to deal with. In the end, features are supposed to be implemented in the system.

These features are sometimes not limited to one Microservice, but require changes to multiple Microservices. Besides, sometimes more changes have to be introduced into a Microservice than one team can handle. In practice, it has proven best to have one team in charge of a Microservice, but to allow also other teams to modify the Microservice. When a feature requires changes to multiple Microservices, one team can introduce all these changes without having to let changes be prioritized by another team. Besides, multiple teams can work on one Microservice in order to implement a greater number of features. Nevertheless, the team assigned to the Microservice is still in charge. In particular, it has to review all modifications to guide its development.

One Microservice per Team?

By the way, it is not really necessary that one team only implements one Microservice. A team can definitely also implement multiple Microservices. However, it is important that a team has a precisely defined responsibility in the domain architecture. In addition, one domain aspect should be implemented in as few Microservices as possible. It can definitely be desirable to implement smaller Microservices so that one team is in charge of more than one Microservice.

2.4 Conclusion

The discussion in section 2.1 about the size of Microservices focuses rather on the technical structure of the system for defining Microservices. For the distribution according to *Bounded Context* (section 2.2) the domain architecture is the most important aspect. Conway's Law (section 2.3) states that Microservices also have effects on the organization. Only together these aspects give a faithful picture of Microservices. Which of these aspects is the most important depends on the use context of the Microservice-based architectures.

3 Why Microservices?

There is more than one reason to use Microservices. Depending on the context the architectural designs can look completely differently. Therefore, it is not only important to know the advantages, but also to judge their relevance for a concrete project and to implement a fitting architecture.

3.1 Scaling Agility

As mentioned in section 2.3, Microservices can affect the organization. Ideally each Microservice should be developed by one team or at least exactly one team should be in charge of the Microservice.

This provides options for the scaling of agile projects: Normally all teams have to coordinate and concertedly work on features. When each team has its own stream of requirements and can implement them by changes to its own Microservice, the teams can work largely independently of each other on features. This allows to tackle also larger projects in an agile manner. In principle the system is just divided into multiple small projects which each can work independently of each other. Apart from supporting a domain architecture it is also very helpful that Microservices can bring features into production without influencing other Microservices. This allows for a largely independent development. In addition to the independence concerning features, Microservices offer also technical independence: Technology decisions can be limited to individual Microservices. This extends the independence the teams have: They cannot only largely independently implement features, but also make their own independent technology decisions.

Thereby Microservice-based architectures enable the independent development of individual Microservices and therefore facilitate the scaling of agile processes to larger project organizations.

3.2 Migrating Legacy Applications

Work with legacy code is often difficult: Systems grown over time are often badly structured so that it is difficult to get an overview. In addition, the code is often of poor quality, and tests are lacking. Besides, the technological basis if often outdated so that modern approaches cannot be used.

Some of these problems can be solved by changing the approach for modifying the system: Instead of modifying the code of the legacy system, the system is supplemented or partly replaced at its external interfaces by Microservices. The advantage: Instead of editing the badly structured and hard to understand legacy code, this code is practically left untouched and rather supplemented by external systems.

The overall goal is the complete replacement of the legacy system by a multitude of Microservices. However, it is easy to start by supplementing the legacy system by Microservices. This does not require a lot of preparations and can thus easily be tried out. Should Microservices not prove to be a good solution in a specific context, they are also easy to remove again from the system.

The easy integration of Microservices are one reason why Microservices are so interesting. Replacing a legacy system by a multitude of Microservices is often a very useful approach to rapidly benefit in a system from advantages like Continuous Delivery.

3.3 Sustainable Development Speed

Microservice-based architectures distribute a system into multiple independently deployable services. The distribution of a system into Microservices is an important architecture decision. It determines the responsibilities of the components.

In a deployment monolith there is also such an architecture at the beginning. However, there it is often lost over time since it is very easy to incorporate new dependencies into a deployment monolith: It suffices to reference a class somewhere in the code. The architecture of an E-commerce system might define for instance that the order process is to call the billing. In contrast, billing may not call the order process. Dependencies which only go into one direction have the advantage that modules remain changeable. In the example it is possible to change the order process without having to modify billing. However, changing the billing might affect the order process since the order process uses the billing.

While implementing features in the billing process, a developer might after all use functionalities from the order process. Something like that happens easily. Experience shows that this initial dependency is soon followed by additional ones so that it is at some point not possible anymore to go on to develop the two components independently since they are using each other.

In the case of Microservices it is not so easy to use another Microservice just like that. Microservices have each an interface, and it is only possible to use them via their interfaces. This requires to call the interface via technologies such as REST or messaging. This does not happen just by mistake.

If the Microservice intended to be used is developed by another team, it can even be necessary to contact this team. Ultimately, the distribution of architecture into Microservices is relatively stable, and in contrast to deployment monoliths the architecture cannot easily get lost. Of course, similar results can also be accomplished by other measures enforcing architecture integrity. There are for instance architecture tools which alert developers to their transgression of architecture rules e.g. Structure101[4] or Sonargraph[5]. However, in the case of Microservices such measures are already integrated into the system.

Replaceability

Another important characteristic of Microservices is their replaceability: Without much effort a Microservice can be replaced by a new implementation. This solves another problem of legacy systems: When a system cannot be maintained anymore, it is often also impossible to rewrite it as the expenditure would just be too large. However, to replace a Microservices is not very difficult.

Conclusion

Within a Microservice-based system it should also in the long run be easy to implement new features since a Microservice is small. If a Microservice should nevertheless stop to be maintainable at some point, it can be replaced. The architecture of the whole system can be expected to be stable in the long run. Therefore, the long term maintainability of the system can be ensured. In summary, Microservice-based systems promise a lasting good maintainability and changeability of the software system.

3.4 Robustness

In a Microservice-based system there is a high robustness in regards to certain problems - in contrast to deployment monoliths: When a functionality in a deployment monolith uses up a lot of CPU

---

[4] http://structure101.com/

[5] https://www.hello2morrow.com/products/sonargraph

or memory, other modules will be affected. If in the worst case a module causes the system to break down, all other modules will likewise not be available anymore.

A Microservice is a separate process or even an individual virtual machine. Therefore, a problem in one Microservice does not influence another Microservice since the operating system or the virtualization isolates the Microservices from each other.

Nevertheless, Microservices are distributed systems. They run on multiple servers and use the network. Servers and network can fail. Accordingly, a Microservice-based system in its entirety should not be very robust since it is more affected by these dangers.

Therefore, Microservices have to be safeguarded from the failure of other Microservices. This is called "Resilience". Resilience can be implemented in very different ways: When an order process cannot be finished, it might be an option to try again later. When a credit card cannot be verified, it might be a possibility to nevertheless perform the order up to a certain upper limit. What this upper limit is, would have to be decided as part of the requirements of the system.

Resilience allows to make a Microservice-based system very robust. The basis for this is the strict separation in processes or virtual machines.

3.5 Continuous Delivery

Continuous Delivery[6] is an approach where software is regularly brought into production. The basis for this is mainly a largely automated process as illustrated in Fig. 4:

- In the commit phase unit tests and static code analyses are performed.

---

[6] http://continuous-delivery-book.com

- Automated acceptance tests ensure that the software correctly implements features.
- Capacity tests on the other hand check whether the performance is fine and whether the expected load can be handled.
- Manual tests can address new features, but also error-prone areas.
- In the end the software goes into production.

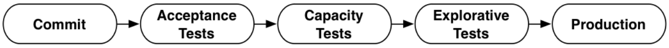

Fig. 4: Continuous Delivery Pipeline

Continuous Delivery is hard to implement in the context of deployment monoliths:

- Automating tests and deployments is complex since deployment monoliths are difficult to bring into production. The database which can be quite large and often contains a lot of data plays for instance a central role. In addition, many third party systems need to be integrated or simulated.
- The tests are laborious. Especially for deployment monoliths modifications can easily have unintended side effects. Therefore a comprehensive regression test has to be performed for each change. This causes a lot of effort and slows the Continuous Delivery Pipeline down.
- Finally, it is hard to safeguard a release. It would be conceivable to create a second environment, to deploy the new version in this environment and only to switch to the new version when it has been tested once more. In such a case it is also possible to fall back to the old version. However, for a deployment monolith such approaches are hard to implement because the required environment is too large and complex.

Microservices are independent deployment units. Therefore, they can have independent Continuous Delivery Pipelines. These Pipelines can be created relatively easily. In addition, it is comparably fast to bring a Microservice through the Continuous Delivery Pipeline into production. Moreover, deployments of Microservices are easier to safeguard. All the problems concerning Continous Delivery in the context of deployment monoliths can be solved by the smaller size of Microservices. Thus Continuous Delivery is profoundly facilitated by Microservices.

Of course some measures are necessary to ensure the independent deployment of Microservices. Nevertheless, the advantages in regards to Continuous Delivery are for many architects and developers an important reason to get interested in Microservices.

3.6 Independent Scalability

Each Microservice runs as an individual process, sometimes even in a separate virtual machine. This allows to scale just the concerned Microservice when a certain functionality is used especially heavily while the other Microservices continue to run with the same capacity.

This does not sound impressive at the start. However, in practice this characteristic of Microservices leads to a number of profound advantages since scaling is facilitated. In general, performance requirements which are limited to certain cases are really demanding. The independent scalability of Microservices allows to concentrate on the ones, which are under heavy load, and to use much less resources to deal with the problem than would be the case for a deployment monolith. This can also be a relevant reason for introducing Microservices.

3.7 Technology Freedom

In principle each Microservice can be implemented in a different technology. Of course, this renders the system overall more complex. This complexity can be limited by the use of standards for operations, monitoring, log formats or deployment. This will ensure at least that the operation is largely uniform.

Still the technology freedom allows for example to use an individual search technology for the product search without requiring extensive coordination with other Microservices and teams. When a team requires a bugfix in a library and therefore wants to use a new version, this change is likewise limited to one team and can therefore be performed by this one team which then carries also the risk. In case of a deployment monolith an extensive coordination would be necessary and accordingly also more tests.

In the end new technologies can be tested without a large migration effort. Risk and expenditure are limited: Initially a single Microservice can be migrated. If the migration does not work, only this one Microservice will fail and, besides, in case of larger problems only this one Microservice has to be newly implemented.

A special project for bringing a deployment monolith to another technology is not necessary for Microservices and the migration is much easier. This entails also other positive consequences: Developers are free to try out new technologies which generally increases motivation.

3.8 Conclusion

Microservices have a large number of advantages. Which advantages are in the end the most important, depends on the concrete context. For many projects the focus is on the replacement of a deployment monolith. In such a case the easy handling of the legacy system (see section 3.2) is an important advantage during migration. Reasons to migrate are in such cases often the wish to scale agile processes (section 3.1) or for an easier implementation of Continuous Delivery (section 3.5).

However, there are also very different scenarios where for instance the objective is to increase the stability of an application in operation. In this case robustness (section 3.4) is an important motivation, and independent scaling (section 3.6) can be another important factor.

Last but not least Microservices promises to make systems maintainable even in the long run (section 3.3) and can use many different technologies where they are most useful (section 3.7).

Therefore, the relevant advantages depend on the respective context. How Microservices should be used in the context of a certain system, likewise depends on the specific advantages which are supposed to be realized.

4 What’s Next?

This primer can only provide a short introduction into Microservices. Therefore, it is an important question how to go on after reading this brochure.

4.1 Microservices: Just a Hype?

Microservices are more than just a hype. Amazon employs the distribution into teams with their own technologies already since 2006. This architecture and this approach is what is nowadays called Microservices. Pioneers like Netflix rightly expected so great advantages from this architecture approach that they were willing to heavily invest into the creation of the necessary infrastructures. Nowadays the technologies they created are available for everybody so that the introduction and the use of Microservices are much easier and less costly.

In addition, the trends to agility, Continuous Delivery and Cloud are reflected by Microservice-based architectures. Even beyond these criteria there are good reasons for Microservices â€“ ranging from individual scalability to robustness. Therefore, Microservices are not only a good supplement for a number of trends, but they represent also a solution for different problems. The trend to use Microservices is thus based on a number of reasons. Therefore it is extremely unlikely that it will just be a short-lived hype.

4.2 Self-contained Systems

An approach based on Microservices are Self-contained Systems[7]. They focus on coarse-grained systems which allow teams to work independently. They should be integrated on the UI level and only use asynchronous communication among each other. This is a more coarse-grained architecture than Microservices. While Microservices can be used in many different scenarios and for many different purposes, as explained in chapter 3, Self-contained Systems represent a more specific approach to Microservices, which is custom-tailored to solve problems of large projects.

4.3 Examples

The primer discusses Microservices only theoretically and does not introduce technologies for implementation.

http://ewolff.com/microservices-demos.html shows several examples for the different options to implement microservices:

Synchronous Communication

There are several options for synchronous communication between microservices:

- The Consul demo[8] is written in Java with Spring Cloud / Boot. The demo uses Consul for service discovery, Apache httpd for routing, Hystrix for resilience and Ribbon for load balancing. It also provides a Prometheus installation[9] for monitoring and an ELK stack[10] for log analysis.

---

[7] http://scs-architecture.org

[8] https://github.com/ewolff/microservice-consul

[9] https://github.com/ewolff/microservice-consul#prometheus

[10] https://github.com/ewolff/microservice-consul#elastic-stack

- The Netflix demo[11] uses the Netflix stack. The demo is written in Java with Spring Cloud / Boot. It uses Netflix Eureka for service discovery, Netflix Zuul for routing, Hystrix for resilience and Ribbon for load balancing.
- Kubernetes is a system to run Docker containers in a cluster. The Kubernetes demo[12] is written in Java with Spring Cloud / Boot. It uses Kubernetes for service discovery, routing and load balancing. The demo also uses Hystrix for resilience. The code does not depend on Kubernetes.
- Cloud Foundry is a PaaS. It provides an application with an environment to run in. The Cloud Foundry demo[13] is written in Java with Spring Cloud / Boot. Uses Cloud Foundry for deployment, service discovery, routing and load balancing. The demo also uses Hystrix for resilience. The code does not depend on Cloud Foundry.

Asynchronous Communication

Asynchronous communication makes it easier to deal with unreliable networks and services:

- Kafka[14] uses Kafka for communication. Kafka is a message-oriented middleware and allows systems to send messages to one another.
- Atom[15] uses REST / HTTP for asynchronous communication with the Atom format.

UI Integration

UI integration provides very loose coupling:

---

[11] https://github.com/ewolff/microservice

[12] https://github.com/ewolff/microservice-kubernetes

[13] https://github.com/ewolff/microservice-cloudfoundry

[14] https://github.com/ewolff/microservice-kafka

[15] https://github.com/ewolff/microservice-atom

- ESI[16] shows how Edge Side Includes (ESI) can be used to integrate the UI of microservices. On microservice is written in Java with Spring Boot, the other one with Go. The Go microservices is built using multi stage Docker containers.
- jQuery[17] shows how jQuery can be used to integrate the UI of microservices.

The website for each demo explains how the demo can be built and started.

4.4 More literature

Additional literature such as the Microservices Book[18] (there is also a German version[19]) by the same author will be needed to appreciate the full scope of Microservices and to learn how to actually implement them technically.

The free booklet Microservices Recipes[20] gives an overview of technologies that can be used to implement a microservices architecture. It explains the demos mentioned above in more detail and illustrates how the technology work and to provide a foundation for the implementation of a microservices architecture. There is also a German version[21]).

The book Microservices - A Practical Guide[22] contains a more detailed description of technologies for the implementation of microservices. In addition, it contains an introduction to microservices

---

[16] https://github.com/ewolff/SCS-ESI

[17] https://github.com/ewolff/SCS-jQuery

[18] http://microservices-book.com

[19] http://microservices-buch.de

[20] http://practical-microservices.com/recipes.html

[21] http://microservices-praxisbuch.de/rezepte.html

[22] http://practical-microservices.com/

and an overview of technologies for the monitoring of microservices. There is also a German version[23]).

4.5 Final Remark

A final remark: the risk associated with introducing Microservices is very limited: A first Microservice has just to be developed and brought into production. This Microservice can for instance supplement an existing deployment monolith. Should this approach not work out, it is very easy to remove the Microservice again.

---

[23] http://microservices-praxisbuch.de/