Typical Migration Strategies

In this lesson, we'll discuss a few typical migration strategies.

WE'LL COVER THE FOLLOWING

Often **there is a concept** for the final target architecture that the migration should achieve, **but no concrete plan** for the first steps to be taken or for the first microservices to be implemented.

In particular, the small steps into which development can be broken down are a major advantage of microservices. A simple microservice is written quickly. Because of its small size, it is also easy to deploy. And if the microservice should not prove itself, not much effort has gone into the microservice and it can easily be removed again.

In the further course of the project, the new architecture can be implemented step by step, microservice by microservice. In this way, the team will avoid large risks.

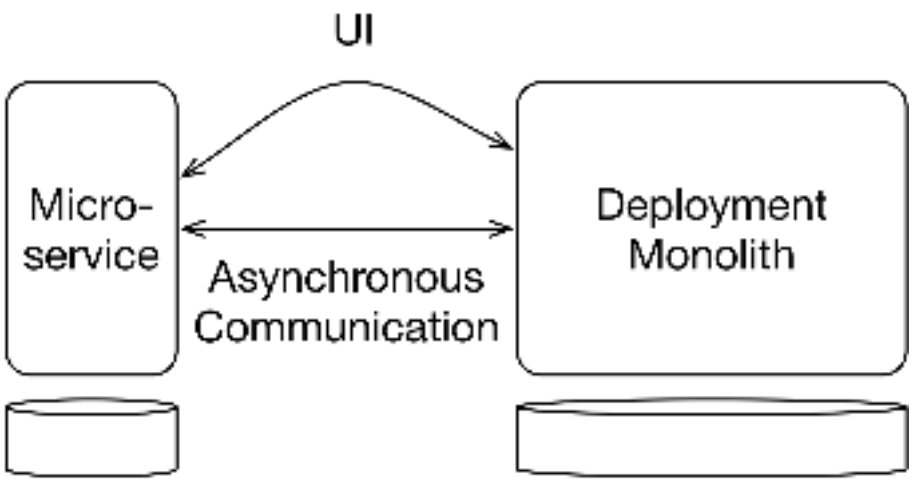
Because the migration process depends on the goals and on the structure of the legacy system, there is no universal approach. Therefore, the strategy presented here must not simply be used as is but must be adapted to the respective situation.

A typical scenario

A typical scenario for a migration to microservices is:

- The aim of the migration is to **increase the development speed**.
    - Microservices need fewer tests for a release and provide easier continuous delivery because they are smaller.
    - Also, the development of individual microservices is quite independent, so less time is spent on coordination. All of this can make development faster.
- The migration to microservices **must provide an advantage in development as quickly as possible**. It makes no sense to invest in an architectural change that only leads to improvement much later down the line.

The migration **strategy proposed** here is **based on extracting individual microservices** in order to achieve an improvement of the situation as quickly as possible. Have a look at the drawing below for a visual.



Approach for a Migration: Asynchronous Communication and UI Integration Between Legacy System and Microservice. The Microservice has its own Data Storage.

Give a preference to asynchronous communication

Integration with the legacy system should take place via asynchronous communication. This **decouples the domain logic of the microservice from the legacy system**.

The legacy system sends events. To do this, the legacy system must be adapted to create events. Since the legacy system is usually poorly maintained, this can be a challenge. Microservices then can decide how to react to these events.

One of the possible benefits is availability: **A failure of the legacy system does not lead to the failure of the microservice**, and a failure of the microservice does not lead to the failure of the legacy system.

Give preference to UI integration

Further integration is conceivable at the UI level. If the legacy system and the microservice are **integrated with each other via links**, then only the URLs are known. What hides behind the URLs can be decided by the linked system and can change without much impact on other systems.

Via links, additional resources can be available. This is the basis of HATEOAS (Hypermedia as the Engine of Application State). The client can interact with the system through the links and does not need to know about possible interaction possibilities.

For example, a link to cancel an order would be sent along with the order. New interaction possibilities can be easily supplemented by new links.

The UI integration also **offers an easy way to operate the microservice and the legacy system in parallel**. Individual requests can be redirected to the microservices, while the remaining requests are still processed by the legacy system.

Often, there is a web server anyway which processes every request and carries out TLS/SSL termination, for example, parallel operation of microservices and the legacy system is then quite simple. The web server only

has to forward each request either to a microservice or to the legacy system.

UI integration is particularly **easy if the legacy system is a web application**. But it is also possible, for example, to integrate web views in a mobile application to thereby integrate parts of the UI as web pages. In such cases, UI integration should really be considered as an option because of its many advantages.

Avoid synchronous communication

Synchronous communication should be used sparingly. It **leads to a close dependence with regard to availability**. If the called system fails, the calling system must be able to deal with this. The degree of coupling in the domain logic is also quite high.

A synchronous call usually describes exactly what needs to be done. Synchronous communication may be necessary if you want the last changes to be visible in the other systems as soon as possible. In a synchronous call, the state at the time of the call is always used, whereas asynchronous communication and replication can lead to a delay until the current state is known everywhere.

Reuse old interfaces?

If there is already an interface, it may be useful to use this interface to **save the effort** of introducing a new interface.

However, the interface **may not be well adapted to the needs of the microservice**. In addition, it **cannot be changed easily** because there are already other systems that also use this interface and are influenced by changes.

The technology the interface uses is not too important for the decision of whether it should be used by a microservice. Microservices can use almost any type of interface. For a migration, it might be a lot easier to use an existing interface even with an awkward technology than to create a new one.

More important are the dependencies the microservices establish: As discussed in this lesson in the previous chapter, the selected pattern for the integration influences the coordination effort and the degree of

integration influences the coordination effort and the degree of

independence. Just reusing an existing interface **might compromise a goal like independent development**.

Integrating authentication

For a system that consists of a legacy system and microservices, the user should have to log in only once. Legacy systems and microservices do not necessarily have the same authentication technologies, but the systems must be integrated in such a way that **a single sign-on is possible**, and the user does not have to log on to the legacy system and microservices separately.

Authentication may also need to provide roles and permissions for authorization in the microservices. Adjustments may also be necessary here.

Replicating data

Even in a migration scenario, **each microservice should have its own database or at least its own database schema**. The goal of the migration is to achieve independent development and simple continuous delivery of the microservices. This is not possible if the microservices and the legacy system use the same database.

A change to the database schema then might have hard-to-predict effects. As a result, the microservice is hardly changeable and difficult to put into production.

Together with asynchronous communication, a separate database means **data replication**. This is the only way for the microservices to implement their own data model. **Changes to the data can be communicated via events**.

Replication should be done in one direction

Replication for a specific part of the data should take place in one direction only. Usually, replication can be done using business events – that is, events that have a meaning for a business expert. One system (a microservice or the legacy system) should trigger the events, and the other system reacts to the events.

So, for example, one system could generate an event like “customer

So, for example, one system could generate an event like "customer registered" and the other system could store the customer data relevant to them. However, there should be only one source of each type of event. Otherwise, it can be very complicated to bring together the changes of the different systems into a consistent state.

Black box migration

Often the code of the deployment monolith is hard to understand and modify. This might even be a reason for migration.

Therefore, it does not make a lot of sense to reverse-engineer the existing code or even restructure it. That way, migrating an existing system requires little or a minimum of knowledge of the system.

Choosing the first microservice for the migration

A legacy system comprises numerous domain functionalities. For deciding about the migration strategy, it can be useful to analyze the domain logic of the legacy system. The result should be a complete and split of the legacy system into bounded contexts.

It is not implemented in the legacy system but can be the goal for the migration to microservices. This analysis can be done without understanding the code. **It is about what the system does, meaning it is enough to treat it as a black box**.

Extracting one of these bounded contexts as a microservice has this advantage: a bounded context is largely independent of other bounded contexts from a domain perspective since it has its own domain model.

However, the question is, **which bounded context should we extract first** from the legacy system? There are a few different approaches.

- To keep the *risk* as low as possible, **an unimportant bounded context with little load** can be the right choice. This makes it possible to gain experience with the challenges of microservices, for example, concerning operation, without taking too great a risk.
- Microservices are meant to simplify development. In order to exploit the advantages of this approach as quickly as possible, you can **migrate a**

advantages of this approach as quickly as possible, you can **migrate a bounded context to a microservice that will have to be changed a lot**

in the foreseeable future. The changes should become easier to introduce after migrating to a microservice, so that the cost of migration quickly pays for itself.

Extreme migration strategy: all changes in microservices

An extreme migration strategy serves to **prevent any changes to the legacy system, and allow only for changes to microservices**.

When a change would have to be made to the legacy system, a new microservice must be created first. The change is then implemented in this microservice instead.

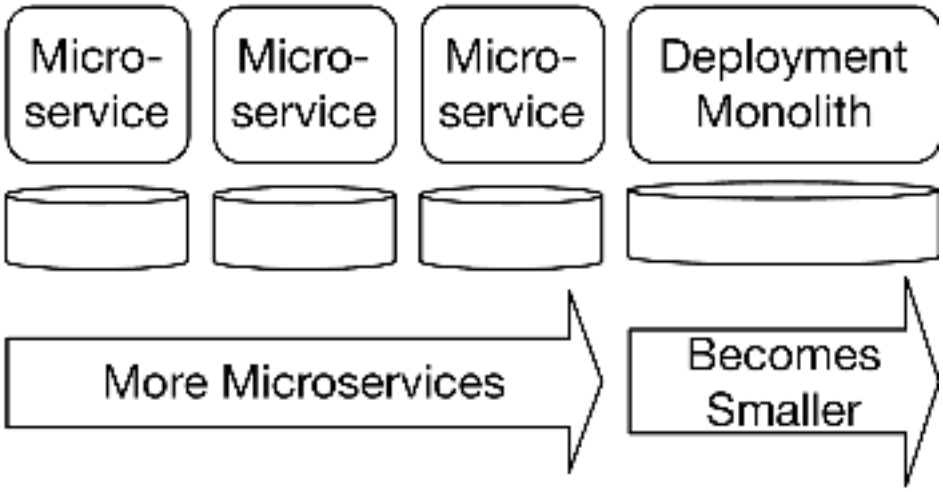
This automatically results in a migration to microservices as more and more logic is implemented in microservices over time. It is very easy to follow this rule.

One problem with this approach is that the microservices are created in random places, namely where the system is currently being changed.

This can result in microservices that implement only parts of a bounded context, whereas other parts of the bounded context are still implemented in the legacy system. Therefore, microservices and legacy systems have numerous dependencies, **making independent development difficult**.

Further procedure: step-by-step migration

The **legacy system can be gradually replaced by microservices**. In the course of the migration, the focus should be on converting parts of the system into microservices that are currently undergoing major changes so that the migration to microservices is worthwhile. This is called the **strangler pattern**. The microservices increasingly strangle the legacy system until nothing is left of the legacy system anymore. Have a look at the drawing below for a visual.



Further Migration: An Increasing Number of Microservices Take over Functionalities from the Legacy System

The full migration to microservices **can take a very long time**. However, this is not a problem: **only those parts of the system are migrated for which migration brings an advantage**.

For example, if a part of the system must be changed, it is migrated to a microservice. That makes the changes much easier. Parts that are never or very seldom changed will be migrated later or even never.

The total time the full migration takes is a result of the flexibility to migrate only what is actually needed. The **migration stops if there is no longer anything worth migrating*. In the end, it makes no sense to invest in the optimization of system parts that are rarely or never changed.

It might even happen that the legacy system could, in theory, be completely migrated, but still retained. Retaining the legacy system can be the best solution when hardly any changes to the legacy system are necessary because all parts that require changes have been migrated to microservices.

QUIZ

1 In what circumstances can synchronous communication be preferable?

In the next lesson, we'll discuss some alternative migration strategies.