Introduction

This lesson gives an overview of why you should take this course.

WE'LL COVER THE FOLLOWING

- Why this course?
- Basic principles
- Concepts
- Recipes
- Source code

Why this course?

Microservices are one of the most important software architecture trends. This course provides a detailed guide to microservices concepts. It lays the foundation for the reader to learn microservices technologies.

Basic principles

To become familiar with microservices, an introduction into microservices-based architectures and their benefits, disadvantages, and variations is essential. However, **this course** explains the basic principles only to the extent required for understanding the **practical implementations**.

Concepts

Microservices require **solutions for different challenges**. Among those are concepts for **integration** (*frontend integration, synchronous and asynchronous microservices*) and for **operation** (*monitoring, log analysis, tracing*). Microservices platforms such as *PaaS* or *Kubernetes* represent exhaustive solutions for the operation of microservices.

Recipes

This course uses **recipes as a metaphor for the technologies**, which can be used to implement the different concepts. Each approach shares a number of features with a recipe.

- Each recipe is described in *practical* terms, including an example technical implementation. The most important aspect of the examples is their *simplicity*. Each example can be easily followed, extended, and modified.
- The course provides the reader a *plethora of recipes*. The readers have to *select* a specific recipe from this collection for their projects, akin to a cook who has to select a recipe for her or his menu. The course shows different options. In practice, nearly every project has to be dealt with differently. The recipes build the basis for this.
- *Variations* exist for each recipe. After all, a recipe can be cooked in many different ways. This is also true for the technologies described in this course. Sometimes the variations are very simple, so that they can be immediately implemented as *experiments* in an executable example.

Source code

Sample code is provided in this course. If the reader wants to really understand the technologies they should understand how the concepts are actually implemented.

In the next lesson, we'll discuss the structure of this course, whether it is the right fit for you, and we'll acknowledge some people who helped make this course happen!

Structure of the Course

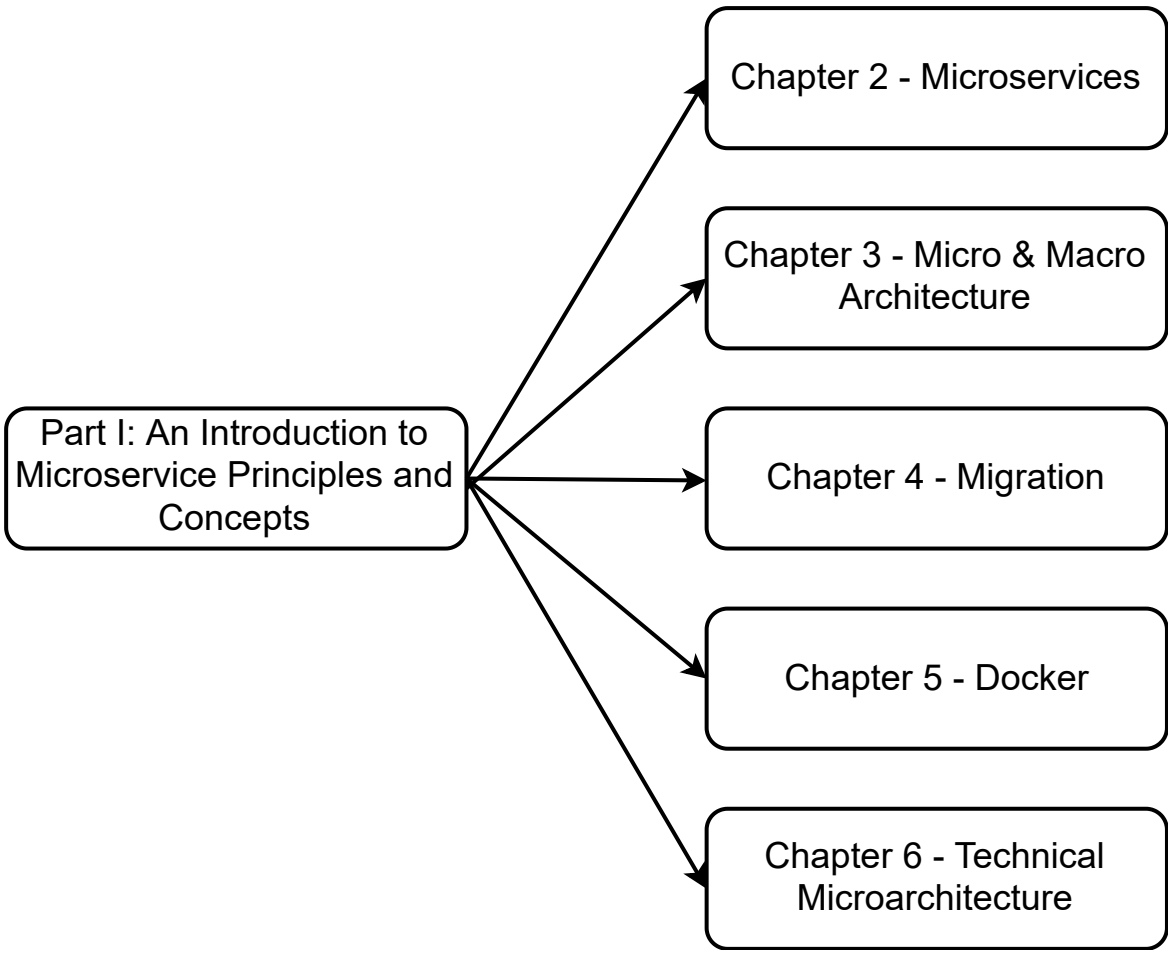
In this lesson, we'll go over the structure of this course!

WE'LL COVER THE FOLLOWING

- Course structure
- Target groups
- Prior knowledge
- Quick start
- Acknowledgements

Course structure

This course is **part I** of a series of courses on microservices to introduce the basic principles of microservices-based architecture and a few important technologies. It’s one thing to define an architecture, and quite another to implement it. Hence, this course presents two technologies for the implementation of microservices and highlights the associated benefits and disadvantages.



Overview of this course

- Chapter 2 **defines the term *microservice***.
- Microservices architecture has two levels: **micro and macro architecture**. They represent global and local decisions as explained in chapter 3.
- Old systems are often supposed to be **migrated into microservices**, a topic covered in chapter 4.
- **Docker serves as the basis for many microservices architectures**. It facilitates the roll-out of software and the operation of the services and is discussed in chapter 5).
- The **technical micro architecture** describes technologies for implementing microservices and is looked at in (chapter 6) .

Target groups

This course explains basic principles and technical aspects of microservices.

Thus, it might be interesting for different audiences.

- For *developers*, this course explains the basic principles of architecture concepts.
- For *architects*, it contains fundamental knowledge about microservices.
- For experts in *DevOps* and *operations*, the recipes in this course provide background information about the concepts behind the microservices architecture approach.
- *Managers* are presented with an overview of the advantages and specific challenges of the microservices architecture approach.

Prior knowledge

This course assumes the reader has some **basic knowledge of software architecture and software development**. All practical examples are documented in such a way that they can be executed with **very little prior knowledge**. This course focuses on technologies that can be employed for microservices using different programming languages. However, the **examples are written in Java** using the Spring Boot and Spring Cloud frameworks so any changes to the code require knowledge of Java.

Quick start

This course focuses primarily on introducing microservices concepts. We will use an example e-commerce system throughout the course to illustrate these concepts.

Acknowledgements

I would like to thank everybody who discussed microservices with me, who inquired about them, or worked with me on this course. Unfortunately, these folks are far too numerous to name individually. The exchange of ideas is enormously helpful and also fun!

Many of the ideas and their implementation would not have been possible without my colleagues at INNOQ. I would especially like to thank Alexander Heusingfeld, Christian Stettler, Christine Koppelt, Daniel Westheide, Gerald

Preissler, Hanna Prinz, Jörg Müller, Lucas Dohmen, Marc Giersch, Michael

Simons, Michael Vitz, Philipp Neugebauer, Simon Kölsch, Sophie Kuna, Stefan Lauer, and Tammo van Lessen.

Also, Merten Driemeyer and Olcay Tümce provided important feedback.

Finally, I would like to thank my friends and family, whom I may have neglected while writing this course – especially my wife. She also did the translation into English.

Of course, my thanks goes out to the people who developed the technologies which I introduce in this course and thereby created the foundation for microservices.

I would also like to thank the developers of the tools of https://www.softcover.io/ and Leanpub.

Introduction

In this lesson, we'll look at an overview of what to expect from this chapter!

WE'LL COVER THE FOLLOWING

- Microservices: definition
  - Advantages of this microservice definition
  - Deployment monolith
  - Size of a microservice
- Chapter walkthrough

Microservices: definition

Unfortunately, there is no universally acknowledged definition for the term *microservice*. In the context of this course the following definition will be used:

> **Microservices** are independently deployable modules.

For example, an **e-commerce system** can be divided into modules for:

- ordering
- registration
- product search

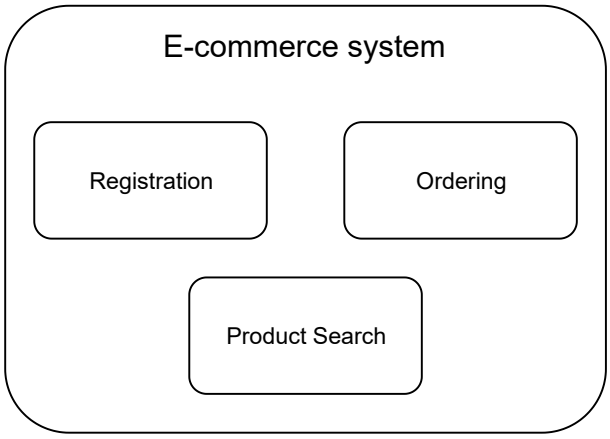
Normally, all of these modules would be implemented together in one application. In this case, **a change in one of the modules** can only be brought

application. In this case, **a change in one of the modules** can only be brought into production by bringing a **new version of the entire application** with all its modules into production. However, when the modules are implemented as microservices, the ordering process cannot only be **changed independently of the other modules**, but it can even be brought into production independently.

This speeds up deployment and reduces the number of necessary tests since only a single module needs to be deployed. Due to this greater level of decoupling, a large project can turn into a number of smaller projects. Each project is in charge of an individual microservice.

To achieve this at the technical level, **every microservice has to be an independent process**. A better solution for decoupling microservices is to provide an independent virtual machine or Docker container for each microservice.

In that case, a deployment will replace the Docker container of an individual microservice with a new Docker container, which starts the new version and its direct requests. The other microservices will not be affected if such an approach is used.



An e-commerce system can be divided into modules as above

Advantages of this microservice definition

The definition of microservices as independently deployable modules has several advantages:

- It is very *compact*.

- It is very *general* and covers all kinds of systems which are commonly denoted as microservices.
- The definition is based on *modules* and is thus a well-understood concept. This allows us to adopt many ideas concerning modularization. This definition also highlights that microservices are part of a larger system and cannot function entirely on their own. Microservices have to be integrated with other microservices.
- The independent deployment is a feature that creates numerous advantages and is therefore very important. Thus, the definition, in spite of its brevity, explains what the most *essential feature* of a microservice really is.

Deployment monolith

A system that is not made up of microservices can only be deployed in its entirety. Therefore, it is called a *deployment monolith*. Of course, a deployment monolith can be divided into modules. The term *deployment monolith* does not make a statement about the internal structure of the system.

Size of a microservice

The above definition of microservices does not say anything about the size of a microservice. Of course, the term *microservice* suggests that especially small services are meant. However, in practice, **microservices can vary hugely in size**. Some microservices keep an entire team busy, while others comprise only a few hundred lines of code. Thus, the size of microservices is ill-suited to be part of the definition.

Chapter walkthrough

This chapter introduces *microservices* and discusses:

- Advantages and disadvantages of microservices to enable the reader to evaluate the applicability and usefulness of this architecture for a specific project.
- The discussion of benefits explains which problems microservices can solve and how this architecture can be adapted for different scenarios.
- The discussion of disadvantages illustrates where technical challenges

and risks lie and how these can be addressed.

- Recognizing advantages and disadvantages is critical for technology and architecture decisions since those have to be aimed at maximizing benefits and reducing disadvantages.

QUIZ

1

A microservice should not be any longer than a few hundred lines of code.

COMPLETED 0%

1 of 3

In the next lesson, let's discuss the advantages of using microservices.

Advantages

There are a number of reasons why we should use microservices. Let's discuss them in this lesson.

WE'LL COVER THE FOLLOWING

- Microservices for scaling development
- Replacing legacy systems
- Sustainable development
  - Replaceability of microservices
  - Dependencies have to be managed
    - In Classical Architectures
    - In the microservices architecture:

Microservices for scaling development

One reason for the use of microservices is the **easy scalability of development**. Large teams often have to work together on complex projects. With the help of microservices, the projects can be divided into smaller units that can work independently of each other.

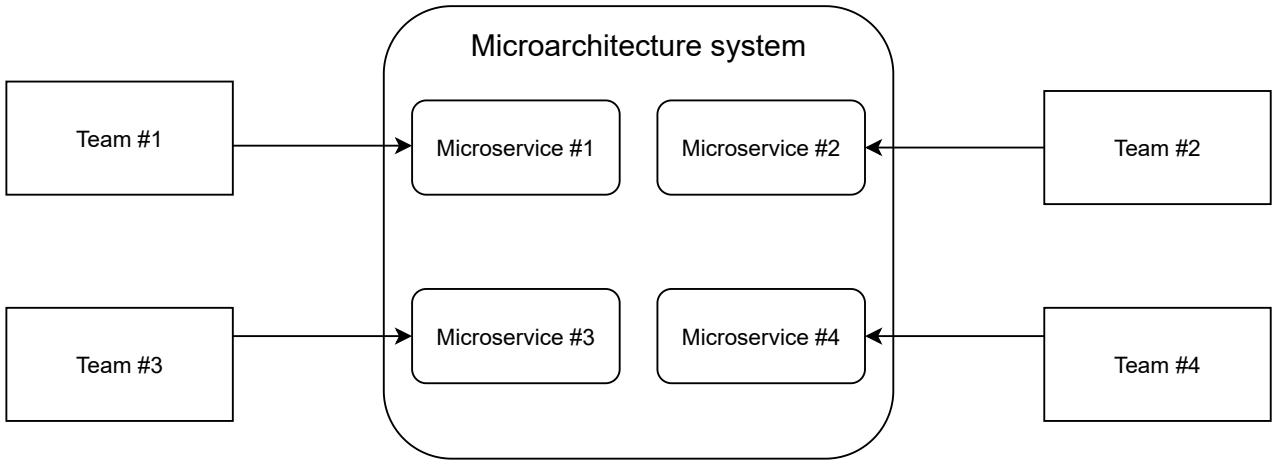
- For example, the **teams** responsible for an individual microservice **can make most technology decisions on their own**.
- When the microservices are delivered as Docker containers, **each Docker container only has to offer an interface** for the other containers.
- The **internal structure of the containers does not matter** as long as the interface is present and functions correctly. Therefore, it is irrelevant which programming language a microservice is written in. Consequently, the responsible team can make such decisions on their own. Of course, the selection of programming languages can be restricted in order to avoid increased complexity. However, even if the choice of the

programming language in a project has been restricted, a team can still

independently use an updated library with a bug fix for their microservice.

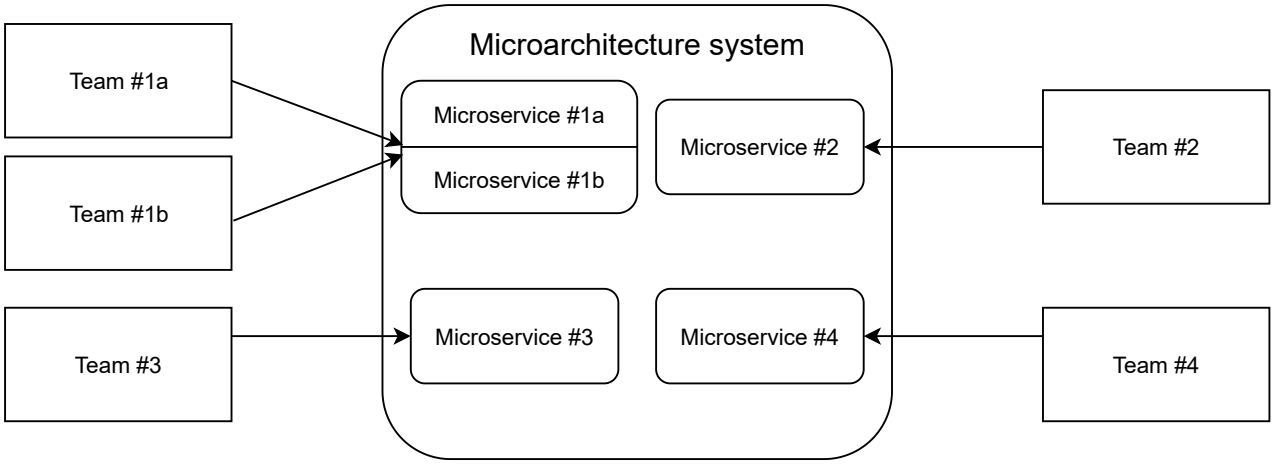
- As stated in the last lesson, when a new feature only requires changes in one microservice, it **can not only be developed independently, but it can also be brought into production on its own**. This allows the teams to work on features completely independently.

Thus, with the help of microservices, **teams can act independently regarding domain logic and technology**. This **minimizes the coordination effort** required for large projects.



In a microarchitecture system, each microservice has a team working on it.

1 of 2



Hence, one mircroservice and consequently the teams working on it can be scaled up

2 of 2

Replacing legacy systems

The maintenance of a legacy system is frequently a challenge since:

1. The code is often badly structured.
2. The changes are not checked by tests.
3. Developers might have to deal with outdated technologies.

Microservices help when working with legacy systems since the existing code does not necessarily have to be changed. Instead, **new microservices can replace parts of the old system**. This requires integration between the old system and the new microservices, for example, via data replication, REST, messaging, or at the level of UI. Besides, problems such as a uniform single sign-on for the old system and the new microservices have to be solved.

But then the microservices are very much like a greenfield project. **No pre-existing codebase has to be used**. In addition, developers can employ a completely different technology stack. This immensely facilitates work compared to having to modify the legacy code itself.

Sustainable development

Microservice-based architectures promise that **systems remain maintainable** even in the long run.

Replaceability of microservices

An important reason for this is the **replaceability of microservices**. When a microservice can no longer be maintained, it can be rewritten. Compared to changing a deployment monolith, this entails less effort because the microservices are much smaller.

However, it is difficult to replace a microservice, on which numerous other microservices depend since changes might affect the other microservices. Thus, to achieve replaceability, **the dependencies between microservices have to be managed appropriately**.

Replaceability is a great strength of microservices. Many developers work on replacing legacy systems. However, when a new system is designed, the

replacing legacy systems. However, when a new system is designed, the

question of how to replace this system after it has turned into a legacy system is rarely asked. Microservices with their replaceability provide an answer.

Hence, individual microservices remain maintainable. If the code of a microservice is unmaintainable, it can just be replaced and it would not influence any of the other microservices.

Dependencies have to be managed

To achieve maintainability, the dependencies between the microservices have to be managed in the long term.

In Classical Architectures

- **Classical architectures often have difficulties** at this level. A developer writes new code and unintentionally introduces a new dependency between two modules, which had been forbidden in the architecture.
- Typically, the mistake goes unnoticed because attention is only paid to the code level of the system and not to the architectural level.
- Often, it is not immediately clear which module a class belongs to. So it is also unclear to which module the developer just introduced a dependency.
- In this manner, more and more dependencies are introduced over time. The originally designed architecture becomes more violated, culminating in a completely unstructured system.

In the microservices architecture:

- Microservices have **clear boundaries due to their interface** irrespective of whether the interface is implemented as a REST interface or via messaging.
- When a developer introduces a new dependency on such an interface, they will notice this because the interface has to be called appropriately. For this reason, **it is *unlikely* that architecture violations will occur** at the level of dependencies between microservices.
- The interfaces between microservices are in a way **architecture**

firewalls since they prevent architecture violations. The concept of architecture firewalls is also implemented by architecture management tools like Sonargraph, Structure101, or jQAssistant. Advanced module concepts can also, generate such a firewall. In the Java world, OSGi limits access and visibility between modules. Access can even be restricted to individual packages or classes.

The architecture at the level of dependencies between microservices also remains maintainable. **Developers cannot unintentionally add dependencies** between microservices. Therefore, microservices can **ensure a high architecture quality** in the long term both inside each microservice and between the microservices.

Thus, microservices enable sustainable development where the speed of change does not decline over time.

QUIZ

Q

Why is it NOT likely that a developer will introduce a new dependency between two modules in a microservice architecture?

COMPLETED 0%

1 of 1

In the next lesson, we'll continue our discussion of the advantages of

In the next lesson, we'll continue our discussion of the advantages of microservices.

Advantage: Continuous Delivery

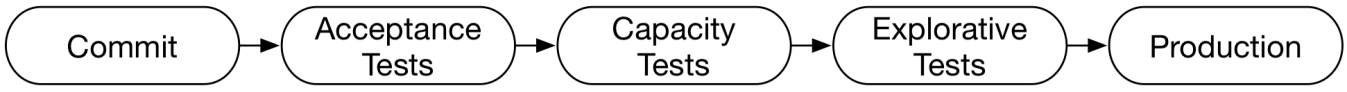
In this lesson, we'll focus on continuous delivery as an advantage of using microservices.

WE'LL COVER THE FOLLOWING

- Continuous delivery
  - Phases
  - Microservices facilitate continuous delivery
  - Deployment must be automated

Continuous delivery

Continuous delivery is an approach where software is continuously brought into production with the help of a continuous delivery pipeline. The pipeline brings the software into production via different phases. Have a look at the following drawing:



Microservices continuous delivery pipeline

Let’s discuss each phase shown above.

Phases

- Typically, the **software compilation**, **unit tests**, and **static code analysis** are performed in the **commit phase**.
- In the **acceptance test** phase, **automated tests** assure the correctness of the software regarding domain logic.
- **Capacity tests** check the performance at the expected load.
- **Explorative** tests serve to perform not-yet-considered tests or to examine new functionalities. In this manner, explorative tests can analyze aspects

that are not yet covered by automated tests.
- In the end, the software is brought into **production**.

Microservices represent independently deployable modules. Therefore each microservice has its own continuous delivery pipeline.

This facilitates continuous delivery.

Microservices facilitate continuous delivery

- The continuous delivery pipeline is **significantly faster** because the deployment units are smaller. Consequently, deployment is faster.
- Continuous delivery pipelines contain many test stages. The software has to be deployed in each stage. Faster deployments speed up the tests and therefore the pipeline.
- The **tests are also faster** because they need to cover fewer functionalities. Only the features in the individual microservice have to be tested, whereas in the case of a deployment monolith, the entire functionality has to be tested due to possible regressions.
- **Building up a continuous delivery pipeline is easier** for microservices. Setting up an environment for a deployment monolith is complicated. Most of the time, powerful servers are required. In addition, third-party systems are frequently necessary for tests. A microservice requires less powerful hardware. Besides, not many third-party systems are needed in the test environments.
    - However, running all microservices together in one integration test can cancel out this advantage. An environment suitable for running all microservices would require powerful hardware as well as integration with all third-party systems.
- The **deployment of a microservice poses a smaller risk** than the deployment of a deployment monolith. In the case of a deployment monolith, the entire system is deployed anew, and in the case of a microservice, only one module. This causes fewer problems since less of the functionality is being changed.

In summary, **microservices facilitate continuous delivery**. Even their support of continuous delivery can be reason enough to migrate a deployment

monolith to microservices.

Deployment must be automated

However, note that microservice architectures can only work when the deployment is automated! Microservices substantially increase the number of deployable units compared to a deployment monolith. This is only feasible when the deployment processes are automated.

Independent deployment means that the continuous delivery pipelines have to be completely independent. Integration tests conflict with this independence. They introduce dependencies between the continuous delivery pipelines of the individual microservices. Therefore, **integration tests must be reduced** to the minimum. Depending on the type of communication, there are different approaches to achieve this for synchronous and asynchronous communication.

QUIZ

1 In what phase of continuous delivery would the performance of an application be checked against the expected load?

COMPLETED 0%

1 of 2

In the next lesson, we'll continue our discussion of the advantages of microservices.

More on Advantages

In this lesson, we'll continue our discussion of the advantages of microservices.

WE'LL COVER THE FOLLOWING

- Robustness
- Independent scaling
- Free technology choice
- Security
- In general: isolation

Robustness

Microservice systems are more robust.

When a memory leak exists in a microservice, only this microservice is affected and crashes. The other microservices keep running. Of course, they have to compensate for the failure of the crashed microservice; this is called **resilience**.

To achieve resilience, microservices can cache values and use them in case of a problem. Alternatively, there might be a fallback to a simplified algorithm.

Without resilience, the **availability of a microservice system might be a problem**. It is likely that a microservice will fail for any reason.

- For example, due to the distribution into several processes, many more servers are involved in the system. Each of these servers can potentially fail.
- Communication between microservices occurs via the network, which can also fail. Therefore, microservices need to implement resilience to achieve robustness.

Independent scaling

Most of the time, **scaling the whole system is not required**. For example, for a shop system during Christmas, the catalog might be the most critical and hardware-consuming part. By scaling the complete system, the hardware is spent on parts that don't require more power.

Each microservice can be independently scaled. It is possible to start additional instances of a microservice and distribute the load of the microservice into the instances. This can improve the scalability of a system significantly.

So, in the previous example, just the catalog would need to be scaled up. For this to work, the microservices naturally have to fulfill **certain requirements**. For example, they must be stateless. Otherwise, requests of a specific client cannot be transferred to another instance, because this instance then would not have the state specific to that client.

It can be difficult to start more instances of a deployment monolith due to the required hardware. Besides, building up an environment for a deployment monolith can be complex. This can require additional services or a complex infrastructure with databases and additional software components.

In the case of a microservice, **the scaling can be more fine-grained** so that normally fewer additional services are necessary and the basic requirements are less complex.

Free technology choice

- Each microservice can be implemented with an individual technology. This facilitates the migration to a new technology since each microservice can be migrated individually.
- In addition, it is simpler and less risky to gain experience with new technologies since they can initially be used for only a single microservice before they are employed in several microservices.

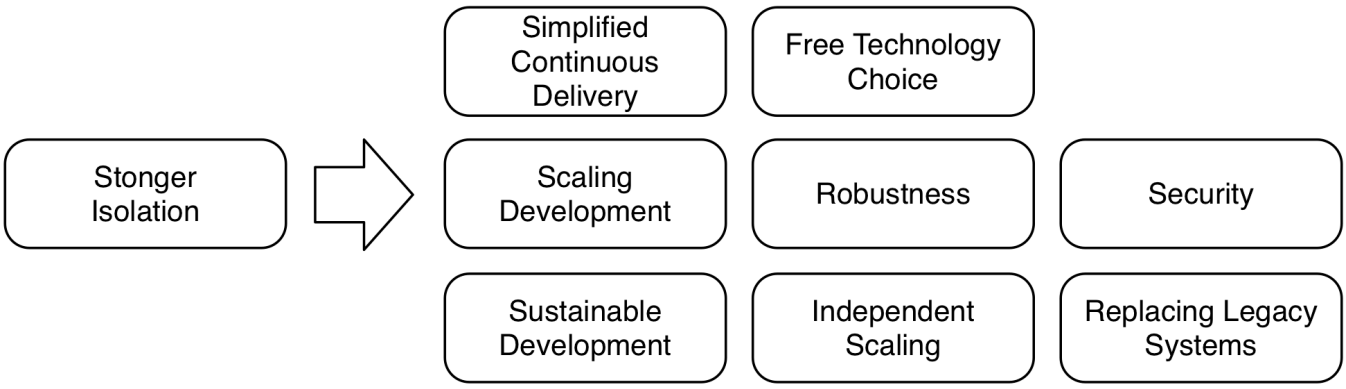
Security

Microservices can be isolated from each other.

- For example, it is possible to **introduce firewalls** into the communication between microservices.
- Besides, the **communication** between microservices **can be encrypted** to guarantee that the communication really originates from another microservice and is authentic. This prevents the corruption of additional microservices if a hacker takes over one microservice.

In general: isolation

In the end, many advantages of microservices can be traced back to a **stronger isolation**.



Isolation as the Source of the Advantages of Microservices

To sum it up:

- Microservices can be deployed in isolation, which **facilitates continuous delivery**.
- They are isolated in respect to failures, which **improves robustness**.
- The same is true for **scalability**. Each microservice can be scaled independently of the other microservices.
- The employed technologies can be chosen for each microservice in isolation, which allows for **free technology choice**.
- The microservices are isolated in such a way that they can only communicate via the network. Therefore, **communication can be safeguarded by firewalls**, which **increases security**.
- Due to this strong isolation, the boundaries between modules cannot be violated by mistake. The **architecture is rarely violated**; this safeguards the architecture.

- In isolation, a microservice can be **replaced with a new microservice**. This enables the low-risk replacement of microservices and allows one to keep the architecture of the individual microservices clean. Thus, isolation facilitates the long-term maintainability of the software.
- **Decoupling** is an important feature of modules. With their isolation, microservices push it to the extremes. Modules are normally only decoupled in regard to code changes and architecture. The decoupling between microservices goes far beyond that. Thanks to decoupling, microservices are smaller. This serves many purposes:
    - Makes it easier to reason about them
    - The security of a microservice is easier to verify
    - The performance is easier to measure
    - It is easier to figure out whether they work correctly
    - That makes the design and also the development easier

QUIZ

1 In a microservice architecture, what will happen if one microservice crashes?

COMPLETED 0%

1 of 3

In the next lesson, we'll look at a few trade-offs of using the microservice architecture, how to prioritize its advantages based on your application, the two levels of microservices and how many microservices can be expected per system.

Tradeoffs, Prioritizing Advantages, & Levels

In this lesson, we'll look at some ways we can prioritize advantages of the microservice architecture and some potential trade-offs that should be considered.

WE'LL COVER THE FOLLOWING

- Prioritizing advantages
- Microservices involve trade-offs
- Two levels of microservices: Domain and technical
- Typical numbers of microservices in a system

Prioritizing advantages

Which of the discussed reasons for switching to microservices is the most important **depends on the individual scenario**. The use of microservices in a greenfield system is the one exception.

More often, a deployment monolith is replaced by a microservice system (see chapter 4). In that case, different advantages are relevant.

- The easier **scaling of development** can be an important reason for the introduction of microservices in such a scenario. Often, it is impossible to work quickly enough with a large number of developers on a single deployment monolith.
- The **easy migration** away from the legacy deployment monolith facilitates the introduction of microservices in such a scenario.
- **Continuous delivery** is often an additional goal. The aim is to increase the speed and reliability with which changes can be brought into production.

The scaling of development is not the only scenario for a migration. For example, when a **single Scrum team** wants to implement a system with

microservices, **scaling development would not be a sensible reason** since the organization of development is not large enough for this. However, **other reasons are possible**. Continuous delivery, technical reasons like robustness, independent scaling, free technology choice, or sustainable development all play a role in such a scenario.

In the end, it is important to **focus on increasing the business value**. Depending on the scenario, an advantage in one of the previously mentioned areas might make the company more profitable or competitive, for example, faster time to market or better reliability of the system.

Microservices involve trade-offs

Depending on the aims, a team can compromise when implementing microservices.

- For example, when **robustness is the goal** of introducing microservices, the microservices have to be implemented as separate Docker containers.
  - Each Docker container can crash without affecting the other ones.
- **If robustness does not matter**, other alternatives can be considered. For example, multiple microservices can run together as Java web applications in one Java application server. In this case, they all run in one process and therefore are not isolated in respect to robustness. Still they are independently deployable.
  - A memory leak in any of the microservices will cause them all to fail.
  - However, such a solution is easier to operate and therefore might be the better trade-off in the end.

Two levels of microservices: Domain and technical

The technical and organizational advantages point to **two levels** at which a system can be divided into microservices.

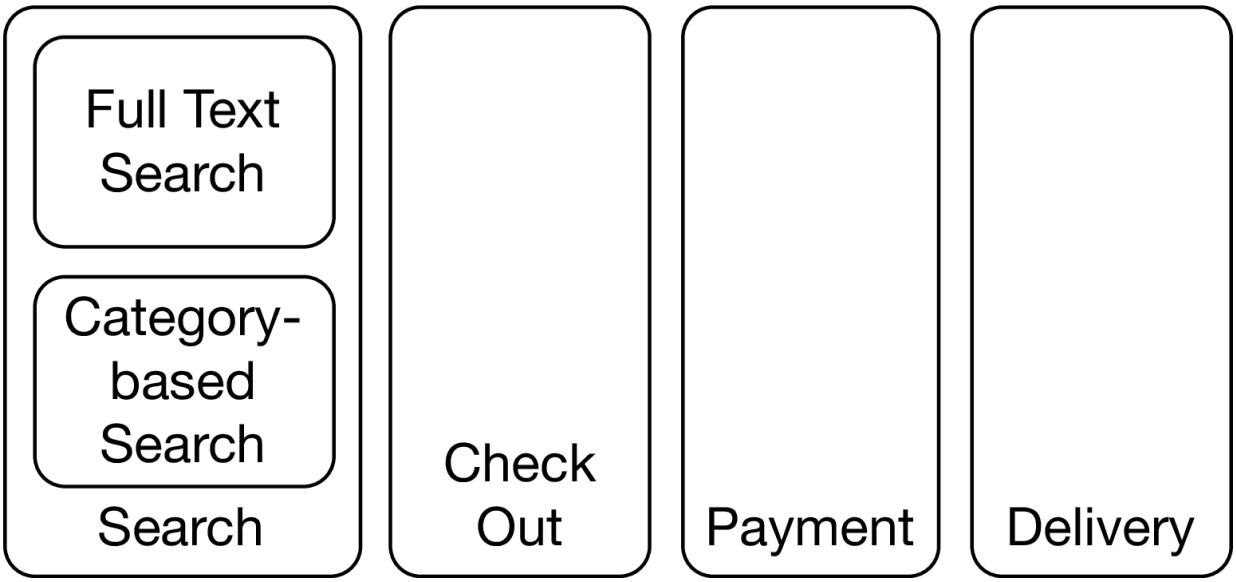
- A **coarse-grained division by domain** enables the teams to develop largely independently and allows them to roll out a new feature with the deployment of a single microservice if it concerns just this one domain, as it usually does. However, sometimes multiple domains are involved and

more than one microservice must be deployed.

    - For example, in an e-commerce system, customer registration and the order process can be examples of such coarse-grained microservices.

- For **technical reasons** some microservices can be *further divided*. These microservices can then be scaled independently of the other microservices.

    - When, for example, the last step of the order process is under especially high load, this last step can be implemented in a separate microservice. The microservice belongs to the domain of the order process, but for technical reasons, it is implemented as a separate microservice. This is an example of a **technical division**.



Two Levels of Microservices

The drawing above shows an example for the two levels. **Based on the domains**, an e-commerce application is **divided into** the microservices:

- Search
    - Full-text search
    - Category-based search
- Check out
- Payment
- Delivery

Search is further subdivided. The full-text search is separated from the category-based search.

- **Independent scaling** can be one reason for this. This architecture allows the system to scale the full-text search independently of the category-based search which is advantageous when both have to deal with different levels of load.
- Another reason could be the **use of different technologies**. The full-text search can be implemented with a full-text search engine, which is unsuitable for a category-based search.

Typical numbers of microservices in a system

It is difficult to state a typical number of microservices per system. Based on the divisions discussed in this chapter, 10-20 coarse-grained domains are usually defined, and each of these might be subdivided into one to three microservices. However, there are also systems with far more microservices.

QUIZ

1 A division by domain always results in the deployment of a single microservice for a change in the system.

COMPLETED 0%

1 of 3

In the next lesson, we'll look at some challenges that the microservice architecture poses.

Challenges

In this lesson, we'll look at possible challenges involved in a microservice architecture.

WE'LL COVER THE FOLLOWING

- Increased operations effort
- Must Be Independently Deployable
- Testing must be independent
- Difficult to change multiple microservices
- Lost overview
- Increased latency and failures
- Weighing benefits and disadvantages
- Experiments

Increased operations effort

- The *operation* of a microservice system **requires more effort than running a deployment monolith**.
  - This is due to the fact that in a microservice system, many more deployable units exist that all have to be deployed and monitored.
  - This is feasible only when the operation is largely automated and the correct functioning of the microservices is guaranteed via appropriate monitoring.

Must Be Independently Deployable

- Microservices have to be **independently deployable**. For example, dividing them, into Docker containers is a prerequisite for this, but it is not enough on its own.
- Changes to interfaces must be implemented in such a way that an independent deployment of individual microservices is still possible.
  - For example, the microservice which implements the interface has

    - For example, the microservice which implements the interface has to offer the new and the old interface. Then this microservice can be deployed without requiring that the calling microservice be deployed at the same time.

Testing must be independent

- Also, **testing must be independent**. When all microservices have to be tested together, one microservice can block the test stage and prevent the deployment of the other microservices making testing much harder.
- Due to the split into microservices, there are more interfaces to test, and **testing has to be independent for both sides of the interface**.

Difficult to change multiple microservices

- Changes that affect **multiple microservices** are **more difficult to implement** than the changes that concern several modules of a deployment monolith.
    - In a microservice system, such changes require several deployments. These deployments must be coordinated.
    - In the case of a deployment monolith, only one deployment would be necessary.

Lost overview

In a microservice system, the *overview* of the microservices can get lost. However, experience teaches that in practice, a sound domain-based division can restrict changes to one or a few microservices. Therefore, the overview of the system is less important because the interaction between the microservices hardly influences development due to the high degree of independence.

Increased latency and failures

Microservices communicate through the *network*. Compared to local communication,

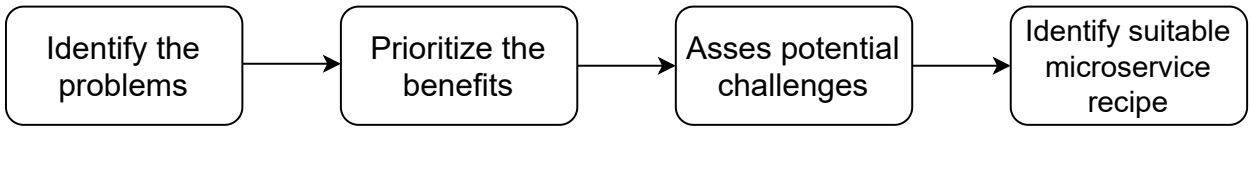
- The **latency is much higher**.
- It is also more likely that **communication will fail**.

A microservices system cannot rely on the availability of other microservices. This makes the systems more complex.

Weighing benefits and disadvantages

The most important rule is that microservices should only be used if they represent the simplest solution in a certain scenario.

The previously mentioned benefits should outweigh disadvantages resulting from the higher level of complexity for deployment and operation. Choosing a more complex solution is rarely a good idea.



Finding the right microservice recipe

Experiments

The following approach helps to find the right recipe to divide a system into microservices.

1. **Identify the problems** in your current system (for example, resilience, development agility, too slow deployment, and so on).
2. For the projects that you've worked on, **prioritize the benefits** of using microservices.
3. Weigh **which of the challenges** in this project could **pose a risk**.
4. Look at the possible technical and architectural solutions in the following chapters to determine the most sensible solutions for their requirements.

For the concrete division into microservices and for technical decisions, additional concepts are necessary. So, let's discuss the question of how best to divide a system into microservices in the next chapter.

QUIZ

QUIZ

1

Suppose you’re designing an application where one microservice gets data from another. If this data fetching fails, the functionality of the app will be compromised. What would be the best course of action in this situation?

COMPLETED 0%

1 of 3

In the next lesson, we’ll look at a quick summary of what we have learned in this chapter.

Chapter Conclusion

That's it for this chapter! Here's a quick summary of what you learned.

Microservices represent an extreme type of modularization. Their separate deployment is the foundation for a **very strong degree of decoupling**. This results in **numerous advantages**.

A crucial benefit is **isolation** at different levels.

- This not only facilitates deployment but also limits potential failures to individual microservices.
- Microservices can be individually scaled, technology decisions affect only individual microservices, and security problems can also be restricted to individual microservices.
- The isolation allows one to more easily develop a microservices system with a large team because it requires less coordination between teams.
- In addition, smaller deployment artifacts make **continuous delivery easier**.
- Moreover, replacing a legacy system is much easier with microservices because new microservices can supplement the system without the necessity of large code changes in the legacy system.

The **challenges** are mostly associated with **operation**. Appropriate technological decisions should strengthen the intended benefits, and at the same time they should minimize disadvantages like the complexity in operation.

Of course, integration and communication between microservices is more complex than the calls between modules within a deployment monolith. The added technological complexity represents an additional important challenge for microservice architectures.

That’s it for this chapter! Next, we’ll study micro and macro architecture.

Introduction

In this lesson, we'll walk through what this chapter holds for us.

WE'LL COVER THE FOLLOWING

- Motivation
- Definition
- Chapter walkthrough

Motivation

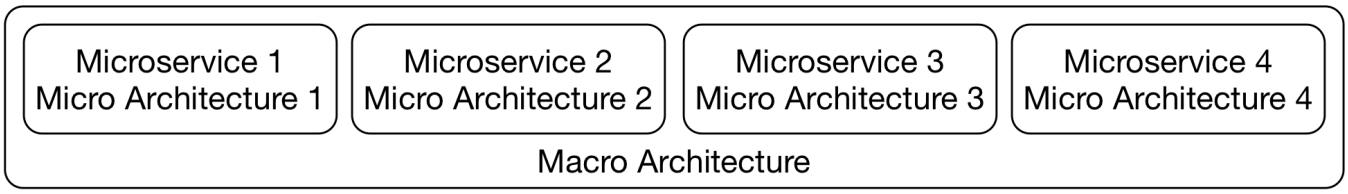
Microservices provide much better decoupling. Therefore, they help to modularize and isolate software modules (see Advantages). However, microservices are modules of a larger system. Therefore, they must be integrated. This poses a challenge for the architecture:

- On the one hand, the architecture has to **ensure that the microservices can work together** to form the overall system.
- On the other hand, the freedom of the **microservices should not be too restricted since this would compromise their isolation** and independence which are required for most of the benefits of a microservice architecture.

Definition

For this reason, it is advisable to divide the architecture into a micro and a macro architecture.

- The **micro architecture** comprises all decisions that can be made individually for each microservice.
- The **macro architecture** consists of all decisions that can be made at a global level and apply to all microservices.



Micro and Macro Architecture

The drawing above illustrates this idea. The overarching **macro** architecture applies to **all microservices**, whereas the **micro** architecture deals with **individual microservices** so that each microservice has its own microarchitecture.

Chapter walkthrough

This chapter illustrates the following:

- The **division of domain logic** into microservices. *Domain-driven design* and *bounded context* are great approaches for such a division.
- The decisions that are part of the *technical micro and macro architecture* and how a **DevOps model** affects these decisions.
- The question of **who** divides the decisions into micro and macro architecture and creates the macro architecture.

QUIZ

1

The e-commerce system discussed in the last chapter, can be divided into microservices like so:

- ordering
- registration
- product search

Suppose the product search team decides to optimize search with a new algorithm. Is this a micro or macro architecture decision?

In the next lesson, we'll look at domain-driven design and an introduction to bounded contexts and strategic design.

Domain-Driven Design & Bounded Contexts

In this lesson, we'll discuss what domain-driven design is and how bounded contexts fit into that definition.

WE'LL COVER THE FOLLOWING

- Bounded context and strategic design
  - An example for a domain architecture
- Domain-driven design: definition
- Bounded context: definition
  - Multiple bounded contexts
- Domain events between bounded contexts
  - Example
- Bounded contexts and microservices
- Evolution

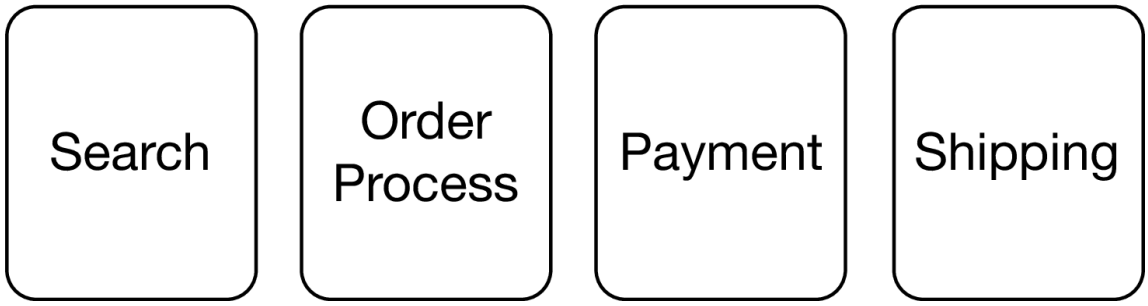
Bounded context and strategic design

Regarding to the domain architecture, the concept of micro and macro architecture has long been a common practice. A macro architecture divides the domains into coarse-grained modules. These modules are further divided as part of the micro architecture.

For example, an e-commerce system can be divided into modules and sub-modules as follows:

- **Customer registration**
- **Order process**
  - Data validation
  - Freight charge calculation
- **Payment**
- **Shipping**

The internal architecture of the **order process** module is, however, hidden from the outside and can be altered without affecting other modules. This **flexibility to change one module without influencing the other** modules is one of the main advantages of modular software development.



Example for a split into multiple domain modules

An example for a domain architecture

The drawing above shows an example of the division of a system into multiple domain modules. In this division, each module has its own domain model. Let's discuss each.

- To **search** successfully, data, such as descriptions, images or prices, must be stored for the products. Important customer data can include, for example, the recommendations that can be determined based on past orders.
- To process orders in the **order process** module, the contents of the shopping cart have to be tracked. For products, only basic information is required such as name and price. Similarly, not too much data concerning the customer is necessary. The most important component of the domain model of this module is the shopping cart. It is then turned into an order that has to be handed over and processed by the other bounded contexts.
- For **payment**, the payment-associated information like credit card numbers has to be kept for each customer.
- For **shipping**, the delivery address is required information about the customer while the size and the weight are necessary information about the product.

This list reflects that **the modules require different domain models**. Not only does the data concerning customer and product differ but so does the entire model and the logic.

Domain-driven design: definition

Domain-driven design (DDD) offers a collection of **patterns** for the domain model of a system. For microservices, the patterns in the area of strategic design are the most interesting. They describe how a domain can be subdivided.

Here are some books you could look into if you are interested in Domain-Driven Design:

- Domain-driven design offers many more patterns that, for example, facilitate the model of individual modules. The original DDD book provides a lot more information. It introduces the term "domain-driven design" and comprehensively describes DDD.
- The more compact book *Domain-driven Design Distilled* focuses on design, bounded context, and domain events.
- The *Domain-Driven Design Reference* is also by the author of the original DDD book. It contains all DDD patterns but without any additional explanation or examples.

Bounded context: definition

Domain-driven design speaks of a **bounded context**. Each domain model is valid only in a bounded context.

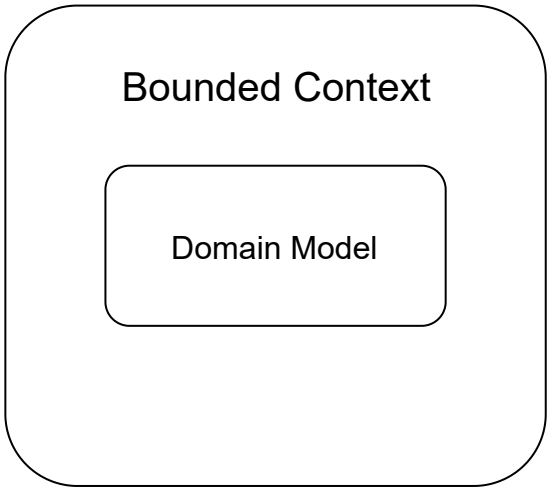
Consequently, *search*, *order process*, *payment*, and *shipping* **are such bounded contexts** because they each have their own domain model.

Multiple bounded contexts

It would be conceivable to implement a domain model that comprises **multiple bounded contexts**. However, such a model would not be the easiest solution.

For example, a price change affects *search*; however, it must not result in a price change for orders that have already been processed in *payment*. It is easier to store only the current price of a product in the bounded context *search*, and to store the price of the product of each order in *payment*, which can also comprise rebates and other complex logic.

Therefore, the simplest design consists of **multiple specialized domain models that are valid only in a certain context**. Each domain model has its own model for business objects such as customers or products.



Each domain model is valid only in a bounded context

Domain events between bounded contexts

For the communication between bounded contexts, we can use **domain events**.

Events can be useful for integrating bounded contexts. Domain events are a part of the domain model as they represent something that happened in the domain. That means they should also be relevant to domain experts.

Example

- Ordering a shopping cart can be modeled as such an event.
- This event is triggered by the bounded context *order process* and is received by the bounded contexts *shipping* and *payment* to initiate shipping and invoicing of the order.

Bounded contexts and microservices

Bounded contexts divide a system by domains. They **do not have to be microservices**. They can also be implemented as modules in a deployment monolith.

If the bounded contexts are implemented as microservices, this results in modules that are independent at the domain and technical level. Therefore, it is sensible to combine the concepts of microservices and bounded contexts.

The dependencies of the bounded contexts as part of strategic designs, as we'll learn in the next lesson, limit this independence. However, since the microservices are part of a larger system, **dependencies between the modules cannot be completely avoided**.

Evolution

There are a number of reasons why new bounded context, and therefore new microservices, might be created:

1. Over time, **new functionalities** might justify **new bounded contexts**.
2. It might become apparent that one bounded context should really be split into two. That might be the case because new logic is added to the bounded context, or the team understands the bounded context better.
3. **New microservices** might be created by dividing a current one due to a **technical reason** (recall division by technicality!).
    - One reason may be to make scalability easier. A microservice may be split in two since the resulting microservices will be smaller and therefore easier to scale. Such reasons might also lead to a larger number of microservices.

QUIZ

1

Suppose you're given the following e-commerce system:

- **Customer registration**

- **Customer registration**
- **Order process**
    - Data validation
    - Freight charge calculation
- **Payment**
- **Shipping**

What will happen to the rest of the modules if the internal architecture of **order process** is changed as follows:

- **Customer registration**
- **Order process**
    - Data validation
    - *Input sanitation*
    - Freight charge calculation
- **Payment**
- **Shipping**

In the next lesson, we'll discuss some strategic design and its key patterns.

Strategic Design & Common Patterns

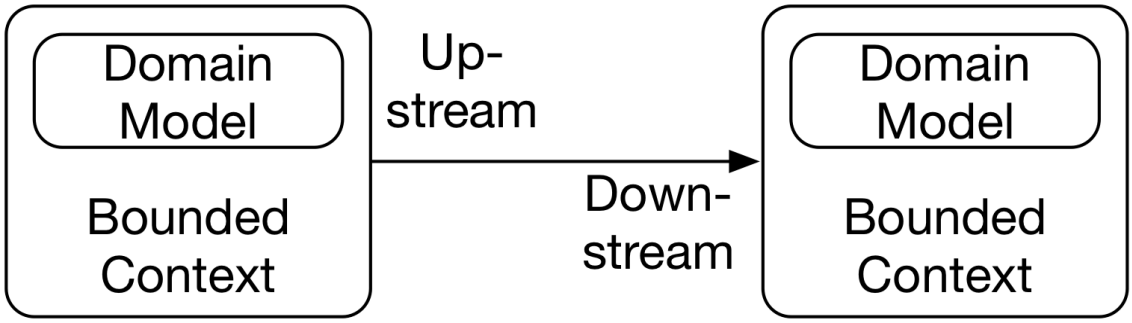
In this lesson, we'll get an introduction to strategic design and look at some important strategic design patterns.

WE'LL COVER THE FOLLOWING

- Strategic design
- The customer/supplier pattern
  - Example
- The conformist pattern
  - Example
- The anti-corruption layer
  - Example
- The separate ways pattern
  - Example
- The shared kernel pattern
  - Example
- The open host service pattern
- The published language model
  - Example
- Selecting patterns
  - Example
    - Tradeoffs to Consider

Strategic design

The division of the system into different bounded contexts is part of **strategic design**, which belongs to the practices of domain-driven design (DDD). The strategic design describes the *integration* of bounded contexts.



Fundamental Terms of Strategic Design

The drawing above shows the fundamental terms of strategic design.

- The **bounded context** is the context in which a specific **domain model** is valid.
- The bounded contexts depend on each other. Usually, each bounded context is implemented by one team.
- The **upstream** team can influence the success of the **downstream** team. However, the downstream team cannot influence the success of the upstream team.
    - For example, the success of the team responsible for payment depends on the order process team.
    - If data such as prices or credit card numbers are not part of the order, it is impossible to do the payment.
    - However, the order process does not depend on the payment to be successful.
    - Therefore, **order processing is upstream**. It can make payment fail. **Payment is downstream** since it cannot make the order process fail.

DDD describes in several patterns how exactly communication takes place. These patterns not only **describe the architecture**, but also the **cooperation *within* the organization**.

The customer/supplier pattern

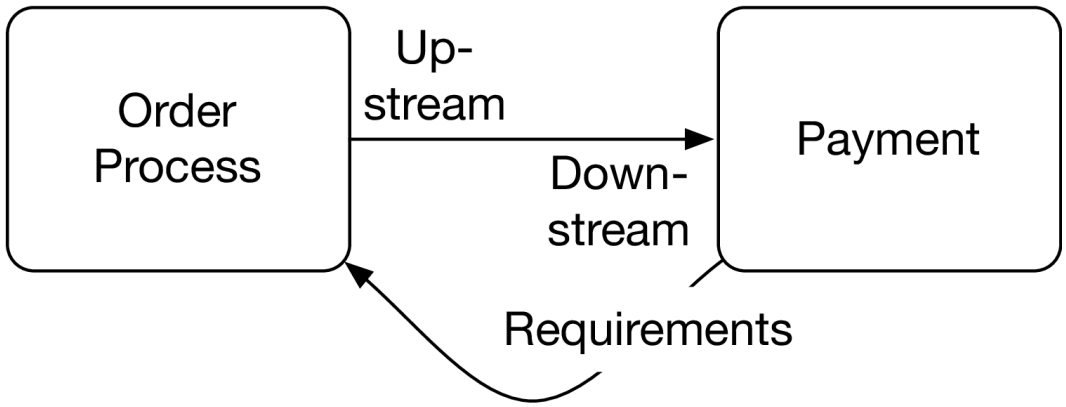
With this *customer/supplier* pattern, the **supplier is upstream** and the **customer is downstream**. However, the customer can factor their priorities

into the planning of the upstream project.

Example

In the drawing below, for example, payment uses the model of the order process. However, payment defines requirements for the order process. Payment can only be done successfully if the order process provides the required data.

So, payment can become a customer of the order process. That way the customer's requirements can be included in the planning of the order process.



Customer / Supplier Pattern

The conformist pattern

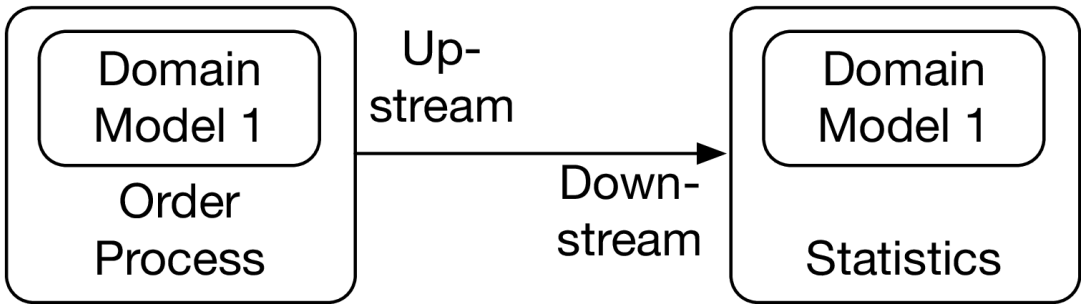
Conformist means that **a bounded context simply uses a domain model from another bounded context**.

Example

In the drawing below, the bounded contexts, **statistics**, and **order process**, both **use the same domain model**. The statistics are part of a data warehouse. They use the domain model of the order process bounded context and extract some information relevant to store in the data warehouse.

However, with the *conformist* pattern, the data warehouse team **does not have a say** in case of changes to the bounded context.

The data warehouse team could not demand additional information from the other bounded context. However, it is still possible that they would receive additional information out of altruism. Essentially, the data warehouse team is not deemed important enough to get a more powerful role.



Conformist: Domain Model Used in Other Bounded Context

The anti-corruption layer

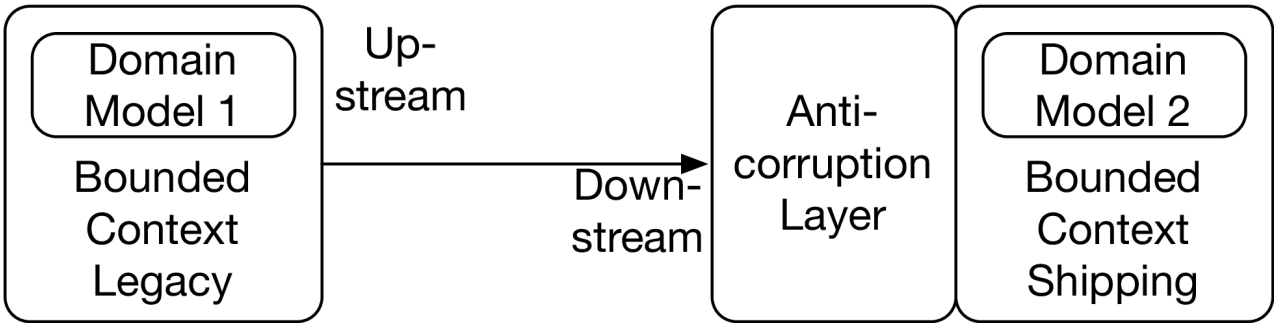
In the case of an **anti-corruption layer (ACL)** pattern, the bounded context does not directly use the domain model of the other bounded context, but it contains a layer for decoupling its own domain model from the model of the bounded context.

This is useful in conjunction with the *conformist* pattern to generate a separate model decoupled from the other model.

Example

The drawing below shows that the bounded context *shipping* uses an ACL at the interface to the bounded context *legacy* so that both bounded contexts have their own independent domain models.

This ensures that the model in the legacy system does not affect the bounded context *shipping*. *Shipping* can implement a clean model in its bounded context.



Anti-corruption Layer with Conformist

The separate ways pattern

With the **separate ways pattern**, the bounded contexts are **not related** at the software level although a relation would be conceivable.

Example

Let's assume that in the e-commerce scenario, a new bounded context, **purchasing**, for the purchase department is added. This bounded context could collect the data for listing products, but it is implemented differently.

With the *separate ways* pattern, the *purchasing* would be separate from the remaining system. When the goods are delivered, a user would use *another bounded context* like *listing* to enter the necessary data and list the products.

The purchasing causes the shipping, which in turn triggers the delivery, and thereby triggers the user to list the product with a different bounded context.

purchasing → shipping → delivery → list product

The shipping of the products is one event in the real world, however, in the software, the systems are separate.

Consequently, the systems are independent and can be evolved completely independently.

Separate Ways

The shared kernel pattern

The Shared Kernel Pattern describes a common core that is shared by multiple bounded contexts.

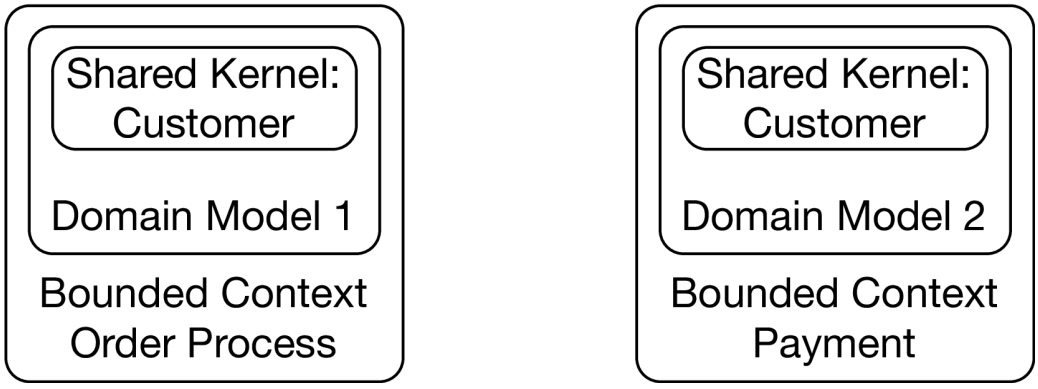
The data of a customer is an example of such a scenario.

However, the *shared kernel* comprises shared business logic and shared database schema and therefore **should not be used in a microservices environment**.

It is an anti-pattern for microservices systems. But because DDD can also be applied to deployment monoliths, there are still scenarios in which a shared kernel makes sense.

Example

In the drawing below the domain model order process and the payment possess a shared kernel.



Shared Kernel

Some patterns are primarily useful in cases where more than one bounded context has to be integrated.

The open host service pattern

Open host service means that the bounded context offers a generic interface with several services. Other bounded contexts can implement their own integration with these services. This pattern is frequently found at public APIs on the Internet. However, it is also a possible alternative within an enterprise.

The published language model

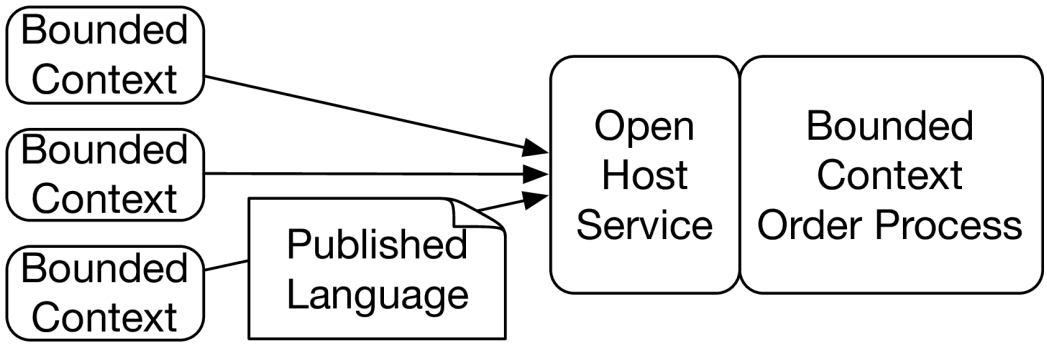
Published language is a domain model accessible by all bounded contexts. For example, this can be a standard format such as EDIFACT for transactions between companies. But it is also possible to define a data structure that is only used inside a company and published, for example, in Wiki.

Example

These models can be used together. The *open host service* can use *published language* for communication. For example, the order process might accept orders from external clients. Providing a specific interface for each external

client is a lot of effort, so there is a generic open host service and a published

language for orders. Each external client can use this interface to submit orders to the bounded context order process.



Open Host Service and Published Language

Selecting patterns

The choice of patterns has to be in line with:

1. The domain
2. The power structures
3. The communication relationships between the teams.

Example

When the bounded context *payment* does not obtain the necessary data from the bounded context *order processing*, the products can be ordered but not paid for. Therefore, the **customer/supplier pattern** is an obvious choice.

However, this is not a fact found in the domain, but rather a **consequence of the power structure**, which in turn depends on the company business model.

Tradeoffs to Consider

Of course, the selected patterns influence the effort necessary for coordination and therefore the degree of isolation between the teams. They set the rules by which the teams must work on the integration.

Thus, a pattern like *customer/supplier* is very desirable as it **requires a lot of coordination**. Still, it might be the right solution depending on domain aspects.

It makes little sense to use a different pattern between *payment* and the *order*

process just to have less coordination. A different pattern might make it impossible for the business to succeed.

QUIZ

1

Consider the following two teams as a part of a small social media app.

1. Image processing. They handle operations on images such as compression and storage.
2. Photo Albums. They handle the look of each user's 'photo albums'.

Which team is upstream and which is downstream in terms of strategic design?

COMPLETED 0%

1 of 4

In the next lesson, we'll discuss architecture decisions.

Architecture Decisions

In this lesson, we'll study some key decisions and at what architecture level, micro or macro, they should be taken.

WE'LL COVER THE FOLLOWING

- Micro and macro architecture decisions
  - Programming languages, frameworks, and infrastructure
  - Database
  - User interface
  - Documentation
- Typical macro architecture decisions
  - Communication protocol
  - Authentication
  - Integration
- Typical micro architecture decisions
  - Authorization
  - Testing
- To Summarize

Microservices provide technological isolation. Therefore, it is possible to extend the concept of micro and macro architecture to **technical decisions**.

For deployment monoliths, these decisions, inevitably, must be implemented globally.

So, only for microservices, technical decisions can be made *within* the framework of macro or micro architecture. However, some decisions have to be part of the macro architecture. Otherwise, the integration will be compromised.

Micro and macro architecture decisions

Decisions can be taken in the context of **either micro or macro architecture**. Let's discuss each.

Programming languages, frameworks, and infrastructure

Programming languages, frameworks, and infrastructure can be defined for each **microservice** individually at the **micro architecture**.

- Then each microservice can also be implemented with a different language.
- Technology, such as the application server, that best suits the specific problems of each microservice can be used.

Programming languages, frameworks, and infrastructure can be defined uniformly for all microservices in the **macro architecture**. This is useful if:

- a company's technology strategy allows only certain technologies
- therefore, only developers with knowledge in certain technologies are hired

Database

At first glance, this decision seems to be comparable to the decision concerning the programming languages, frameworks, and infrastructure but databases are different because:

- They store data.
- The loss of data is usually unacceptable.
    - Therefore, there must be a backup strategy and a disaster recovery strategy for a database.
    - Setting these up for many different databases requires considerable effort.

Micro: Each microservice can also have its own instance of the database. If databases were defined at the **micro architecture**.

- A crash of one database will cause only one microservice to fail which makes the entire app **more robust**.

- However, the **higher effort involved**, especially concerning operation, is an argument against individual instances.

Macro: To avoid needing many different databases, the database can be defined as part of the **macro architecture** for all microservices.

- Even if the database is defined in the macro architecture, **multiple microservices must not share a database schema**. That would contradict the bounded contexts.
- The domain model in the database schema would be used by several microservices. This would couple the microservices too strongly. Even with a unified database, the microservices must have separate schemata in the database.

User interface

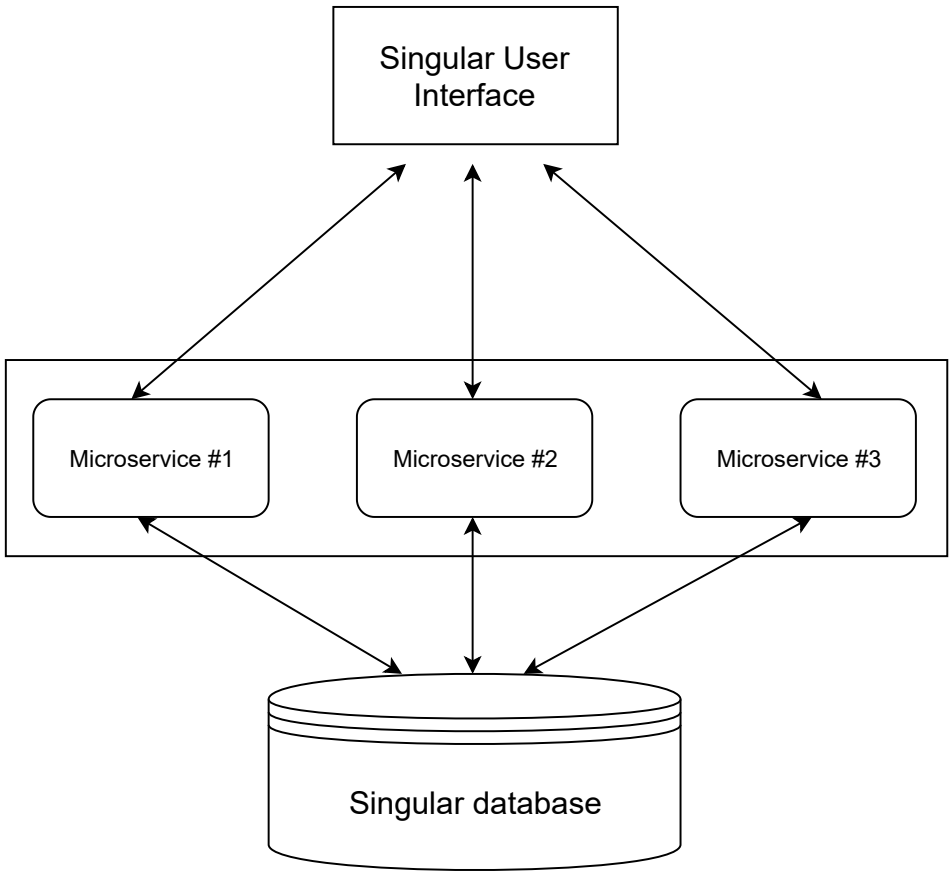
If microservices have their own user interface (UI), the *look and feel* of microservices can be a micro or macro architecture decision.

Micro: sometimes a system has **different types of users** (back office and customers, for example) with **different requirements** for the UI which are often incompatible with a uniform look and feel. A micro architecture decision for the UI is suitable in this case.

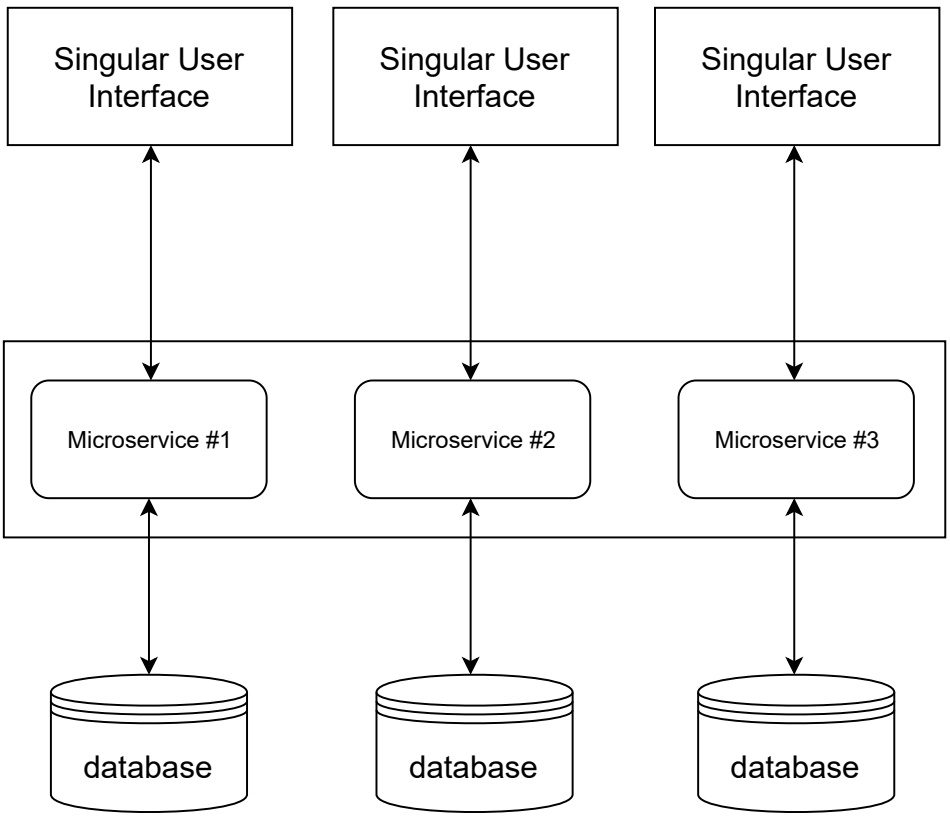
- Often there are concerns that a microservice level decision **will cause inconsistencies** in the look and feel; however, the UI can also **diverge in a monolithic system**. Hence, defining appropriate style guides and artifacts is the only way to achieve a consistent look and feel for large systems, regardless of the use of microservices.

Macro: Often a system should have a uniform UI; therefore, the look and feel must be a macro architecture decision.

- Shared `CSS` and `JavaScript` are often not enough to ensure a common style of the UI of all microservices since uniform technical artifacts can be used to implement very different types of user interfaces. Therefore, a **style guide must become part of the macro architecture**.



The UI and the database are a macro architecture decision



The UI and the database are a micro architecture decision.
Each micro service can provide it's own UI.

Documentation

It may be necessary to standardize the *documentation*.

Micro: The documentation should be part of the micro architecture if the same team will build and maintain the microservice.

- A certain level of documentation makes it easier to later hand the microservice over to another team.
- It may also be necessary to document certain aspects of the microservices in a uniform manner.
    - For example, for security reasons, some systems need to keep track of the libraries used in the microservices. In the case of a security vulnerability in a specific library, it is then possible to identify which microservices need to be fixed.

Macro: Of course, the decision about the documentation can also be part of the macro architecture.

- Standardized documentation can provide an overview of the system and the dependencies between microservices.

Typical macro architecture decisions

There are some decisions that must always be taken at the level of macro architecture. Ultimately, all microservices together should result in a coherent system. This requires some standards.

Communication protocol

The *communication protocol* of the microservices is a typical macro architecture decision.

- Only if all microservices provide a **uniform interface**, for example, a REST interface or a messaging interface, can they communicate with each other coherently.
- In addition, the data format must be **standardized**. It makes a difference

- In addition, the data format must be **standardized**. It makes a difference whether systems communicate with JSON or XML, for example.

If the communication protocol was a **microservice decision**, i.e., a different communication channel between each microservice, **a coherent system will not exist** and will disintegrate into islands that communicate with each other in different ways.

Authentication

With *authentication*, a **user proves their identity**. This can be done with a password and a username, to name a common example.

Since it is unacceptable for the user to re-authenticate with every microservice, the entire microservice system should use a **single authentication system**. The user then enters a username and password once and can then use any microservice.

Integration

Integration testing technology is also a typical macro architecture decision. All microservices must be tested together, so they must run together in an integration test. The macro architecture must define the necessary prerequisites for this.

Typical micro architecture decisions

Certain decisions should be taken for each microservice individually. Therefore, they are typically part of the micro architecture.

Authorization

The *authorization* of the user determines what a user is **allowed to do**. The authorization should be done in the respective microservice.

Which user is allowed to initiate what action, i.e., authorization, is part of the domain logic, and therefore belongs to the microservice like the other domain logic.

If this was decided at the macro architecture, the domain logic would be implemented in a microservice itself, but the decision about **which part** of the domain logic is available to which user would be made centrally, which is

difficult, especially with complex rules.

- For example, if orders up to a certain upper limit can be triggered by certain users, authorization, concrete upper limits, and possible exceptions belong to the microservice **order**.

Authentication assigns the user roles used in authorization.

- For example, a microservice can define which actions a user with the role of *customer* can trigger and which actions a user with the role of *call center agent* can trigger.

Testing

The *testing* can be different for each microservice. Even the tests are ultimately part of the domain logic.

In addition, there may be different non-functional requirements for each microservice.

- For example, one microservice can be particularly performance-critical, whereas another is more safety-critical.

These risks must be covered by an individual focus in the tests.

Since the tests can be different, the **continuous delivery pipeline** is also different for each microservice. It must include the relevant tests. Of course, the technology for the continuous delivery pipeline can be standardized.

- For example, each pipeline can use a tool like `Jenkins`. What happens in the respective pipelines, however, depends on the respective microservice.

Jenkins

To Summarize

The following table shows the typical micro and macro architecture decisions:

Micro or Macro	Micro Architecture	Macro Architecture
Programming Language	Continuous Delivery Pipeline	Communication Protocol
Database	Authorization	Authentication
Look and Feel	Tests of the Microservice in Isolation	Integration Tests
Documentation		

QUIZ

1 Suppose that it has been decided to use a REST interface between microservices for communication. What sort of technical decision could this have been?

COMPLETED 0%

In the next lesson, we'll discuss some factors that influence the operation of applications.

Operation: Micro or Macro Architecture?

In this lesson, we'll discuss some factors that influence the operation of applications.

WE'LL COVER THE FOLLOWING

- Configuration
- Monitoring
- Log Analysis
- Deployment Technology
- Macro architecture operation with separate operations teams
- Standardize only technologies!
- Testing the operation macro architecture
- “You build it, you run it”: operation as micro architecture
- Operation as a whole is micro or macro architecture

Some decisions in the area of micro and macro architecture mostly influence the operation of the applications. Let’s take a look at a few.

Configuration

We must define **the interface with which a microservice obtains its configuration parameters**. For example, a microservice can get these settings via an environment variable or read them from a configuration file. These parameters include both:

- **Technical** parameters such as thread pool sizes
- Parameters for the **domain logic**

The decision of how to store and generate the configuration data is independent of these parameters. The data can be stored in a database, for example. Either configuration files or environment variables can be generated from the data in the database.

generated from the data in the database.

Note that the information on which computer and under which port a microservice can be reached, **does not** belong to the configuration, but to the **service discovery**.

Configuring passwords or certificates is also a challenge that can be solved with other tools. To do this, Vault is a good choice because this information must be stored in a particularly secure way and must be visible to as few employees as possible in order to prevent unauthorized access to production data.

Monitoring

Monitoring is about the **technology that tracks metrics**. Metrics provide information about the state of a system. Examples include the number of requests processed per second or business metrics, such as revenue.

The question of *which* technology is used to track the metrics is independent of which metrics are captured. Every microservice has different metrics because every microservice has different challenges. For example, if a microservice is under a very high load, then performance metrics are useful.

Log Analysis

Log analysis defines a **tool for managing logs**.

Although logs were originally stored in log files, they are now stored on **specialized servers**. This has a few advantages. For example, it makes it **easier to analyze and search the logs**, even with large amounts of data and

many microservices.

In addition, new instances of a microservice can be started when the load increases and can be deleted again after the load decreases. In this case, the logs of this microservice instance should still be available, even if the microservice was deleted long ago due to a decreasing load. If the logs are stored only on a local device, the logs would be gone after the microservice has been deleted.

Deployment Technology

Deployment technology determines **how the microservices are rolled out**. For example, this can be done with Docker images (see chapter 6), Kubernetes Pods, a PaaS, or installation scripts.

These decisions define how a microservice behaves from an operational point of view. Typically, these decisions are either all part of the macro architecture or the micro architecture.

Macro architecture operation with separate operations teams

Whether decisions in the area of operation belong to micro or macro architecture **depends on the organization**.

For example, a team can develop microservices but bear no responsibility for their operation. The operations team is responsible for the operation of all microservices. In this scenario, decisions for operation must be made at the level of macro architecture.

It is generally unacceptable for the operations team to learn a different approach for the operation of each microservice, especially because the number of microservices is much larger than the number of deployment monoliths for the same project.

Another reason for a macro architecture decision on the operation of microservices is that individual solutions in this area bring few advantages. Although a programming language or framework can be more or less suitable for a particular problem, the same applies to a much lesser extent to technologies in the field of operation.

technologies in the field of operation.

Standardize only technologies!

When these decisions are made at the level of macro architecture, **they standardize only the technologies**.

Which configuration parameters, monitoring metrics, log messages, and deployment artifacts of a microservice are a decision at the level of the individual microservice.

The independent deployment must also be retained as a core feature of the microservices. This means that it must be possible to independently change the configuration parameters for each microservice in order to adjust the configuration when a new deployment takes place.

Testing the operation macro architecture

Adherence to the macro architecture rules can be checked with tests. The microservices are deployed in an environment. The tests check whether the rules for uniform deployment are adhered to. The test then verifies whether the microservice delivers metrics and log information in the defined way. Something similar is also possible for configuration.

The test environment for these tests should be very minimalistic and should not contain any other microservices or a database. In this manner, the microservice is tested in an environment in which it cannot possibly work. When such a situation occurs in production, it is particularly important that the microservice provides logs and metrics to analyze potential problems. Similarly, the test also checks the resilience of the microservice.

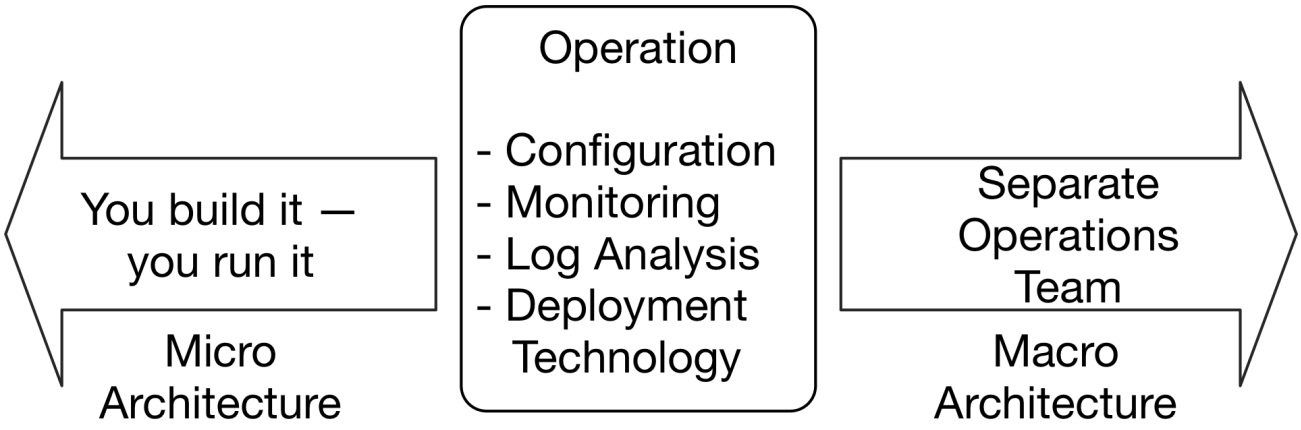
“You build it, you run it”: operation as micro architecture

There is a form of organization in which operational aspects have to be part of the micro architecture. If the same team is to develop and operate the microservice, they must also be able to choose the technology. This approach can be described as “you build it, you run it”. The teams are each responsible for a microservice, for its operation *and* development. You can only expect this level of responsibility from the team if you allow them to choose their

own technologies.

Operation as a whole is micro or macro architecture

Decisions for operation can be taken either at the level of micro or macro architecture. Making operation decisions part of the macro architecture is useful if there is a separate operations team, while a “you build it, you run it” organization must make these decisions at the level of micro architecture. The drawing below illustrates this point.



Depending on the organization, operation is part of the micro or macro architecture

QUIZ

1 Which of the following best describes why storing logs on a specialized log server is adventageous?

In the next lesson, we'll discuss the reasons for making as many decisions as possible at the micro architecture level rather than the macro architecture level.

Give a Preference to Micro Architecture!

There are good reasons for making as many decisions as possible at the micro architecture level rather than at the macro architecture level. Let's discuss each.

WE'LL COVER THE FOLLOWING

- Macro architecture decisions: Best practices and advice
- Evolution of macro architecture

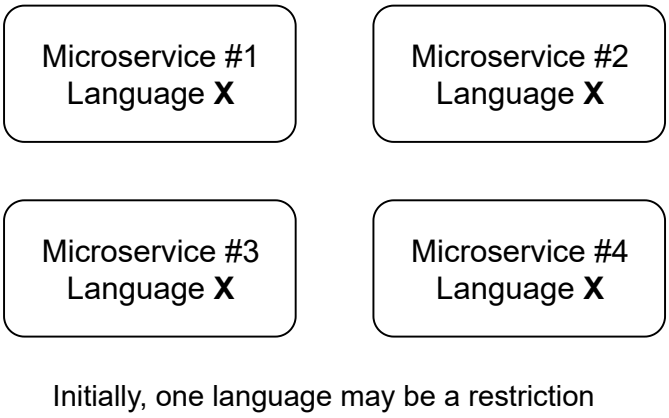
Macro architecture decisions: Best practices and advice

- Specifying only a few points in the macro architecture helps with focusing. Many teams have failed when trying to implement a far-reaching unification in a complex project or IT landscape. If there are few macro architecture rules, the chances increase that the rules are actually successfully implemented.
- The rules should be **minimal**.
    - For example, a macro architecture rule can *define the monitoring technology*. However, it is not necessary to standardize *how* the metrics are measured in the application. After all, it is important only that the metrics are created. How this happens is irrelevant. The macro architecture rule should only define a protocol for transferring metrics but leave the selection of the library for creating and transferring metrics to the micro architecture. In this way, the teams can choose the most appropriate technologies.
- The macro architecture rules have to be **consequently enforced**. For example, when the metrics are not generated, the operations team cannot simply bring the microservice into production. It is important to get rid of all unnecessary macro architecture elements.

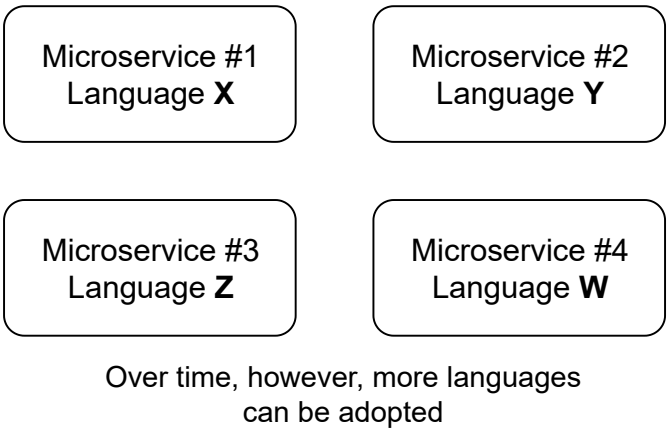
- In addition, **independence** is an important goal of microservices. Too many macro architecture rules run counter to this goal as they hinder the independence of the teams through central control.
- Complying with macro architecture should be in the **self-interest** of the teams responsible for the microservices. Violations of macro architecture usually mean that microservices cannot go into production because operations cannot support them.

In addition to mandatory macro architecture rules, recommendations and best practices are advisable. However, they do not have to be enforced but are optional for every microservice.

In the end, the goal of macro architecture is to create freedom. Advice and references to best practices are therefore good additions.



1 of 2



2 of 2

Evolution of macro architecture

At the beginning of a project, **restrictive rules may initially apply**. For example, a single programming language and a fixed stack of libraries can be defined. This reduces learning effort and operating costs.

Over the duration of the project, **more programming languages and libraries can then be allowed** to introduce modern technologies. This leads to a more heterogeneous system which is certainly preferred over updating all microservices at once because such updates entail a high risk.

QUIZ

1 Which of the following macro architecture rules are better and why?

1. User data for analytics must be stored in a JSON format
2. The technology xyz should be used to gather user data for analytics and the data should be stored in a JSON format

COMPLETED 0%

1 of 3

In the next lesson, we'll discuss organizational aspects!

Organizational Aspects

In this lesson, we'll study some organizational aspects to architectural decisions.

WE'LL COVER THE FOLLOWING

- Uncontrolled growth?
- Who defines macro architecture?
  - Committee of representative team members
  - Independent architecture committee
- How to enforce?
- Testing conformance

There is a connection between decision and responsibility. Whoever makes the decision takes responsibility. Therefore, if the decision about the technology of metrics is made as part of the macro architecture, then the macro architecture group must take responsibility. For example, they would be responsible if this technology proves unsuitable in the end because it does not cope with the amount of data.

If the responsibility for monitoring microservices is completely transferred to the teams, then the teams must also be allowed to select a technology.

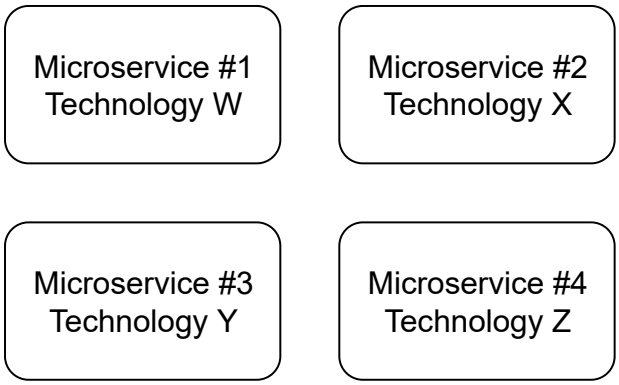
Uncontrolled growth?

The freedom regarding micro architecture can lead to **a huge number of technologies in use**. But this is not necessarily the case. If all teams have had good experiences with a particular monitoring technology, then a new microservice will most likely be monitored with the same tool. Using another tool would require a great deal of effort. Other options are only evaluated and used if the tool used so far is insufficient.

Even without a macro architecture rule, there is standardization if

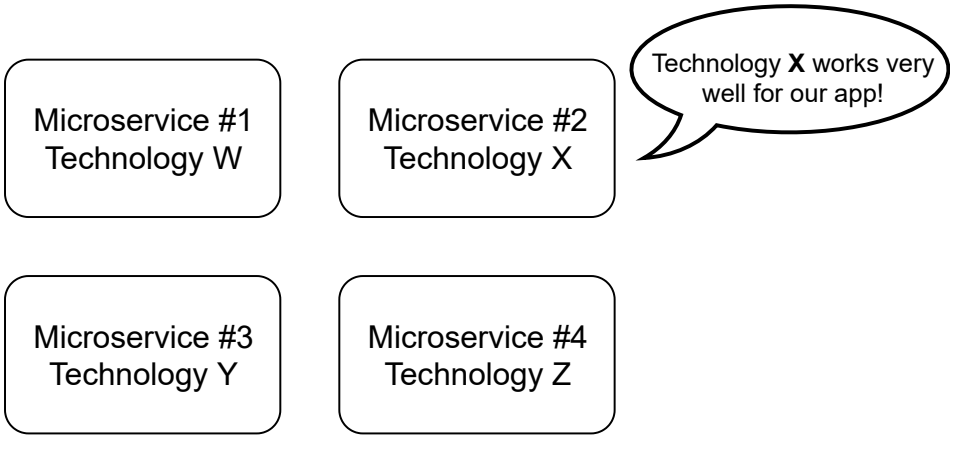
uniform decisions bring advantages for the teams. The prerequisite for this

is, of course, an exchange between the teams about best practices and about which technologies work and which do not.



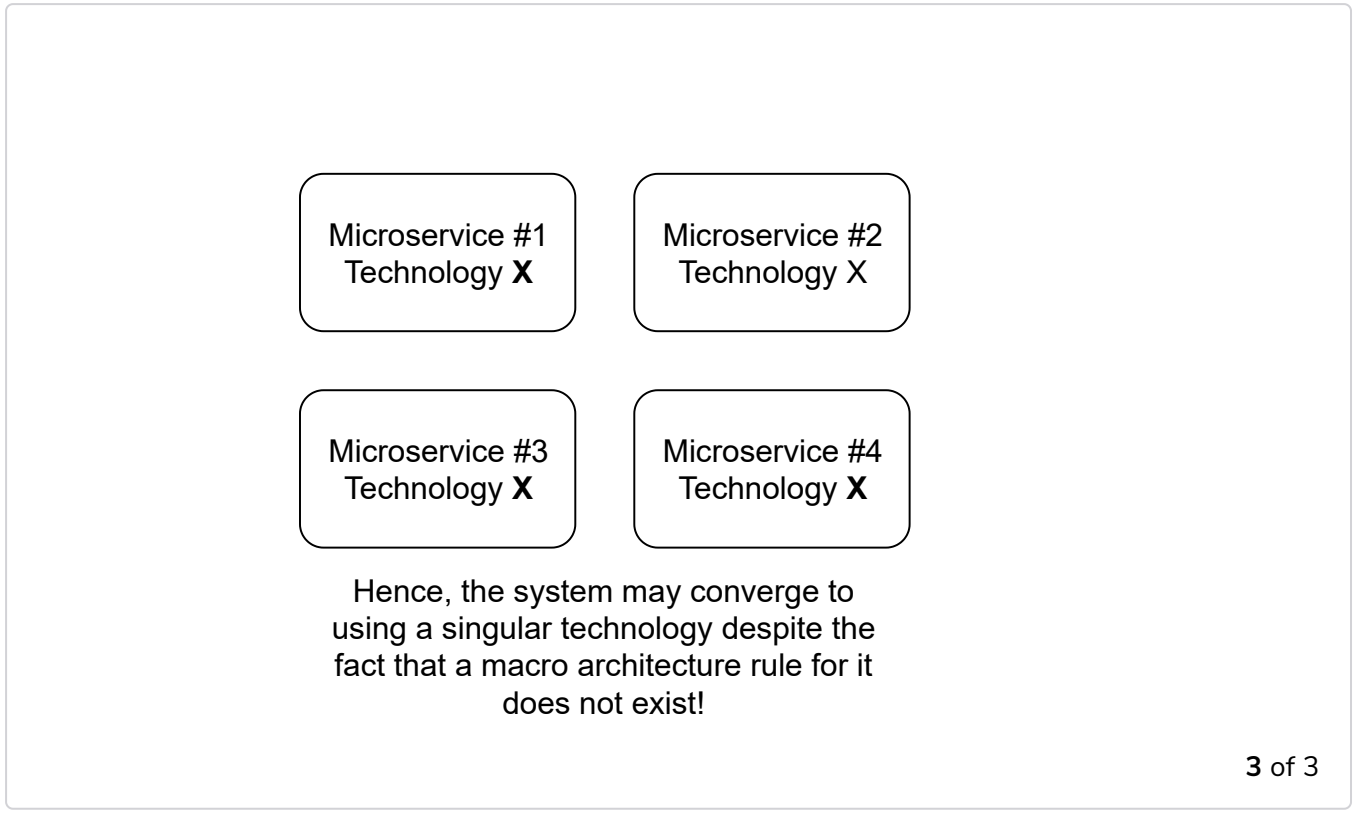
Initially, each microservice may be using a different technology

1 of 3



A team may communicate the effectiveness of a technology to the rest

2 of 3



Who defines macro architecture?

Committee of representative team members

Macro architecture restricts the freedom of the teams when it comes to implementing the microservices. This can be counteracted by having a **committee define the macro architecture**, which consists of one member of each team.

However, it is possible that the committee **may become too large to work effectively**. With ten teams, the team would have ten members and effective work is then hardly possible. You can reduce the number of members by **excluding teams** or sending **individual members as representatives for multiple teams**.

Unfortunately, the team members are often too focused on their own microservices to be interested in the overall picture of macro architecture.

Independent architecture committee

The alternative is to have an **independent architecture committee** decide on macro architecture, which is staffed by architects who do not belong to the

macro architecture, which is staffed by architects who do not belong to the teams.

In such a scenario, it is important that:

- This **body's goal is to support the teams** in their development of microservices and to moderate decisions rather than enforce them. The most important work takes place in the teams. Therefore, the macro architecture should support the teams and not hinder them.
    - Collaboration between the architecture committee and the teams can also be improved by the members of the architecture committee working at least partly in teams.
- The members of the committee are **integrated and interested in the developed system**.
    - The specific domain and business requirements should never be forgotten.
    - An important part of the work on the architecture is to understand stakeholders and ensure that their goals are supported by the architecture.

How to enforce?

The need for macro architecture should be understandable because it ensures that the entire system can be developed and operated. **To enforce** the macro architecture, the team should **document reasons for each rule** to avoid unnecessary discussions behind their reasoning.

For example, certain macro architecture rules may be necessary to allow the operations team to bring the software into production, or to ensure that compliance rules are followed.

It's not so much about enforcing rules as it is about **promoting macro architecture and conveying the ideas/reasons** for macro architecture. If good reasons for changing the macro architecture exist, improving the architecture might be a better option than enforcing an obsolete one.

Testing conformance

In some cases, it is possible to test the conformance to the macro architecture by deploying a microservice and checking its log output and metrics. This

ensures that deployment, logging, and monitoring all conform to the defined macro architecture.

> Such a test is called a **black box test**, which checks the behavior of the microservice from the outside.

The **benefit** of this approach is that it:

- Does not limit the free choice of technology for implementing microservices,
- Does not enforce unnecessary standards for specific frameworks.

Therefore, testing for the conformance on the code level does not make a lot of sense.

QUIZ

1

Even if an operational technology is not standardized at the macro architecture level, certain technologies are naturally conformed to by teams that communicate well.

COMPLETED 0%

1 of 3

In the next lesson, we'll look at Independent Systems Architecture principles.

Independent Systems Architecture Principles

In this lesson, we'll discuss Independent Systems Architecture's nine principles.

WE'LL COVER THE FOLLOWING

- Conditions
- Principle #1: The system must be divided into modules
  - Evaluation
- Principle #2: Two separate levels of architectural decisions
  - Evaluation
- Principle #3: Modules must be separate processes/containers/VMs
  - Evaluation
- Principle #4: Standardized integration & communication
  - Evaluation
- Principle #5: Standardized metadata
  - Evaluation
- Principle #6: Independent continuous delivery pipelines
  - Evaluation
- Principle #7: Operations should be standardized
  - Evaluation
- Principle #8: Standardized interface
  - Evaluation
- Principle #9: Modules have to be resilient
  - Evaluation
- Summary

Micro and macro architecture are fundamental to the idea of microservices. However, it is hard to understand why there should be two levels of architecture.

> **ISA (Independent Systems Architecture)** is the term for a collection of fundamental principles for microservices. It is based on experiences with microservices gained from many different projects.

The name already suggests that these principles aim to build software out of **independent systems**. Macro and micro architecture are very important for this goal.

A minimal macro architecture leaves a lot of freedom to the level of the micro architecture, making the systems independent. Technical decisions can be made for each system without influencing the other systems.

ISA defines the term micro and macro architecture. Also, the principles explain what the minimum requirements for macro and micro architecture are.

Conditions

Must is used for principles when they **absolutely have to be adhered to**.

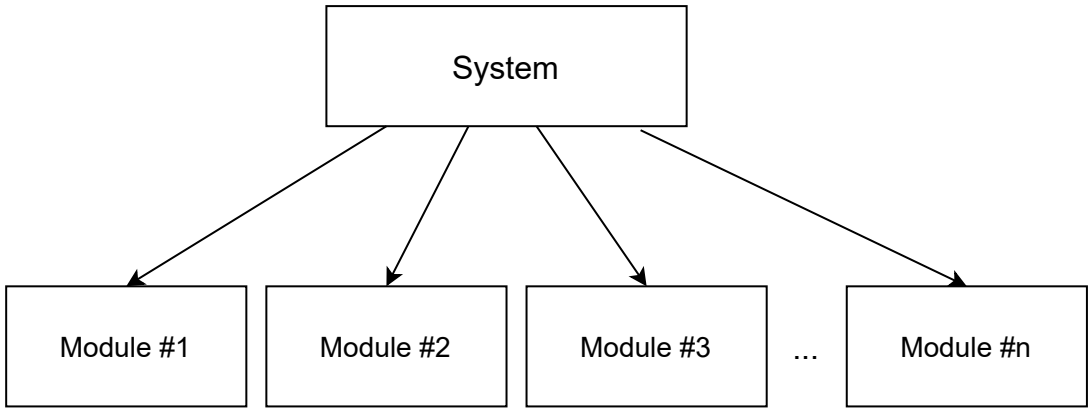
Should describes principles which have many advantages but **do not have to be strictly followed**. We'll now discuss each principle. The **ISA principles** are not only a great guideline for building microservices, but they also explain why macro and micro architecture are so important.

Principle #1: The system must be divided into modules

The system **must be divided into modules** that offer *interfaces*. Accessing modules is only possible via these interfaces. Therefore, modules may not depend directly on the implementation details of another module, such as the data model in a database.

Evaluation

The **first** ISA principle states that a system must be built from modules. This is common knowledge.



System Modularization

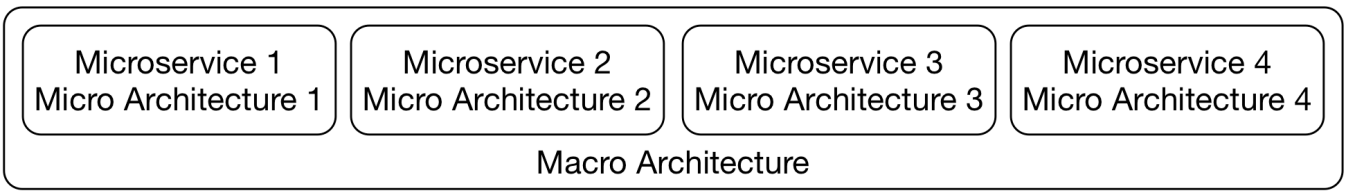
Principle #2: Two separate levels of architectural decisions

The system **must have two clearly separated levels** of architectural decisions:

- **Macro architecture** comprises decisions which concern all modules. All further principles are part of the micro architecture.
- **Micro architecture** comprises those decisions which can be made differently for each individual module.

Evaluation

The **second principle** defines two levels of architecture: macro and micro architecture.



Micro and Macro Architecture

Principle #3: Modules must be separate processes/containers/VMs

Modules **must be separate processes, containers, or virtual machines** to maximize independence.

Evaluation

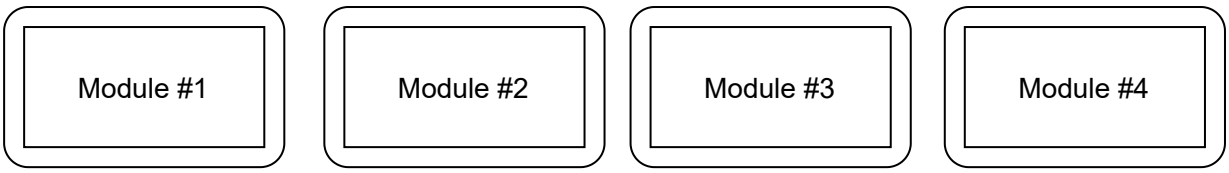
Evaluation

In a deployment monolith, **most of the decisions will be on the macro architecture level**. For example, a deployment will be written in one programming language, so the programming language has to be a decision on the macro architecture level. The same is true for frameworks and most of the other technologies.

To make more decisions on the micro architecture level, each module must be implemented in a separate container as this principle states. ISA says that **the reason** why microservices run in containers **is the additional technological freedom** that cannot be achieved in a deployment monolith. Therefore, microservices add more independence and decoupling to the architecture.

An approach where each microservice is a WAR and all run together in one Java application server does not fit this principle. Actually, the compromise concerning the free choice of technology and the robustness is so high that this approach usually does not make a lot of sense.

Because decoupling is so important, ISA and microservices actually provide fundamental improvements to modularization.



Modules are separate processes/containers/virtual machines

Principle #4: Standardized integration & communication

The **choice of integration and communication options must be limited and standardized** for the system.

- The integration might be done with synchronous or asynchronous communication, and/or on the UI level.
- **Communication must use a limited set of protocols** like RESTful HTTP or messaging. It might make sense to use just one protocol for each integration option.

Evaluation

Although the goal of ISA is to create a minimal macro architecture, **some decisions still need to be made on the macro level**. This is what the rest of the principles explain. As a start, principle four states that integration and communication must be standardized. The last three chapters of this course discuss a few technology stacks for integration and communication.

The decision to use a specific technology for integration and communication influences all modules and must, therefore, be done on the macro architecture level. It is therefore also a very important decision in microservices systems.

Without a common integration approach and communication technology, it is hard to consider the system as a system and not just a few services that can barely communicate with each other.

Principle #5: Standardized metadata

Metadata, for example, for *authentication*, **must be standardized**. Otherwise, the user would need to log in to each microservice.

- This might be done using a token that is transferred with each call/request.
- Other examples might include a trace ID to track a call and its dependent calls through the microservices.

Evaluation

This principle states that metadata for tracing and authentication must be standardized. Such **metadata must be transferred between the microservices** and must, therefore, **also be a part of the macro architecture**. This course does not discuss security aspects of microservices, including metadata for authentication.

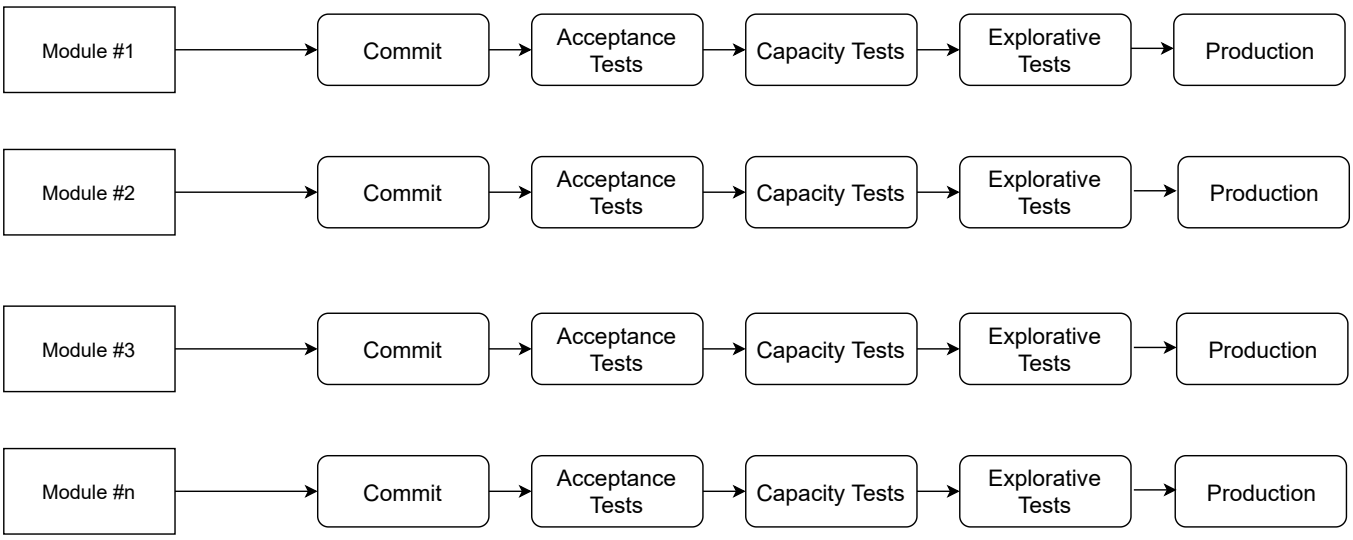
Principle #6: Independent continuous delivery pipelines

Each module **must have its own independent continuous delivery pipeline**. Tests are part of the continuous delivery pipeline; therefore, the tests of the

modules have to be independent, too.

Evaluation

This principle extends the idea of independent deployment as the definition of microservices from the last chapter.



Each module has it's own independent continuous delivery pipeline

Principle #7: Operations should be standardized

Operations should be standardized. There can be exceptions from the standard when a module has very specific requirements. These operations comprise:

- configuration
- deployment
- log analysis
- tracing
- monitoring
- alarms

Evaluation

This principle says that the operations of microservices should be standardized. It is not in all cases necessary to standardize operations. With a separate operations department, standardization is the only way to handle a large number of microservices.

However, with a “you build it – you run it” organization, standards are not necessary as each team operates their microservices. Actually, a standardized operations approach might not fit all microservices. In that case, the teams need to come up with their own operations technologies. A standard may not be useful then.

Principle #8: Standardized interface

Standards for operations, integration, or communication **should be enforced on the interface level**.

- For example, the communication protocol and data structures could be standardized to a specific JSON payload format exchanged using HTTP, but every module should be free to use a different REST library/implementation.

Evaluation

This principle states that standards should only be defined on the interface level. The technologies discussed throughout the last three chapters can be used in this way. They provide interfaces and client libraries for all commonly used programming languages.

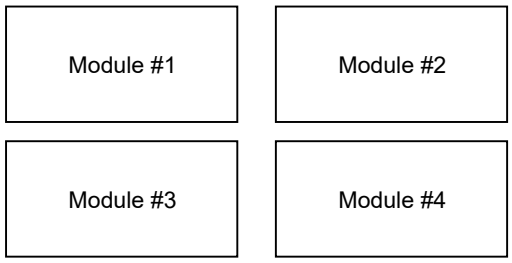
Principle #9: Modules have to be resilient

Modules have to be resilient. This means that:

- They may not fail when other modules are unavailable or when communication problems occur.
- They must be able to shut down without losing data or state.
- It must be possible to move them to other environments (server, networks, configurations and so on) without the module failing.

Evaluation

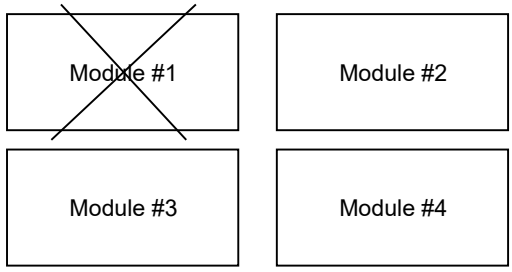
This principle addresses resilience. Asynchronous communication makes resilience easier. If a microservice fails, a message will be transferred later but the failed microservice will not cause another microservice to fail.



4 modules running

Modules must be resilient

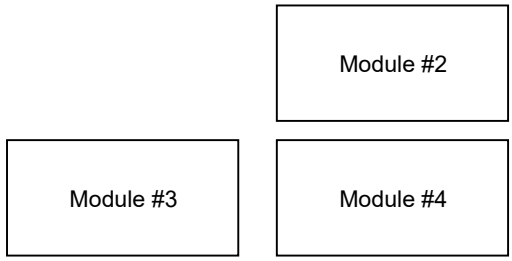
1 of 3



Module #1 fails

Modules must be resilient

2 of 3



The rest keep going

Modules must be resilient

3 of 3

Summary

The ISA principles represent a good summary of the ideas introduced in this chapter – that is, a division between micro and macro architecture as the main benefit of microservices. ISA also explains why the rest of the course

focuses on integration and communication technologies and technologies for

operations. These are the fields that a macro architecture has to cover. Therefore, these decisions are very important and also hard to change because they influence all microservices in the system.

QUIZ

1

The communication method between microservices **has** to be standardized.

COMPLETED 0%

1 of 3

In the next lesson, we'll discuss some variations on what we've discussed so far in this chapter.

Variations

In this lesson, we'll discuss some variations to the approaches discussed in this chapter.

WE'LL COVER THE FOLLOWING

- More complex rules
- Experiments

In the domain macro architecture, strategic design and domain-driven design are ultimately unrivaled as approaches.

However, the bounded contexts depend on the specific project. **Identifying the right bounded contexts is a central challenge** when designing the architecture of a microservices system.

The **technical micro and macro architecture also have to be devised** for each project. This depends on many **factors**:

- Organizational aspects such as *DevOps organization* or having a separate operations team has an influence.
- In addition, *strategic technology decisions* can play a role.
- Even the *hiring policy* can be a factor. Eventually, there have to be experts available who can work in the teams to manage the technologies.

More complex rules

In reality, the rules of micro and macro architecture are **often more complex**.

For example, **a whitelist can exist for the programming language**. In addition, there can be a procedure for adding more programming languages to the whitelist, for example, via a committee. And finally, there can be a general limitation to programming languages that run on the JVM (Java

Virtual Machine).

Such a rule **has elements of a macro architecture decision**. There is a whitelist and a restriction to JVM languages. **At the same time, it also has micro architecture elements**. After all, a team can select one of the programming languages from the whitelist and even extend the whitelist.

Therefore, rules are often in place for every point in practice that allow the teams and microservices a certain amount of leeway. These rules are **not purely micro or macro architecture rules** but lie somewhere in between.

Experiments

The approach for defining micro and macro architecture can look like this:

- Consider a project you are familiar with. Look at its domain model and **consider the following questions**:
    - Would a division into multiple domain models and bounded contexts make the system easier?
    - In how many bounded contexts would you split the system? Typical projects consist of about ten bounded contexts. However, the exact number will vary for each individual project.
    - Determine the use cases which the system implements. Group use cases and analyze whether these use cases can be addressed by a domain model. By doing so, these use cases form a bounded context in which the domain model is valid.
    - Is a further division for technical reasons sensible? The technical reasons can comprise independent scalability or security (see also “Two Levels of Microservices” in this lesson).
- **Define for your project whether the individual decision should be part of micro or macro architecture**.
- **Work out at least one of the decisions in more detail**. For example, there could be a whitelist of programming languages, or only one programming language might be allowed that can be used by all microservices, or even a procedure for extending the whitelist.

QUIZ

Q

Suppose a whitelist for the databases is defined at the macro level. Which of the following is true concerning the decision to use a specific database?

COMPLETED 0%

1 of 1

In the next lesson, we'll look at the chapter conclusion.

Chapter Conclusion

In this lesson, we'll formally conclude this chapter with some notes regarding decisions at the micro and macro architectural level.

WE'LL COVER THE FOLLOWING

- Micro architectural decisions
- Macro architectural decisions

Micro architectural decisions

If decisions can actually be **different for each microservice**, then they are part of the **micro architecture**.

Decisions at the micro architecture level are better suited to the self-organization of teams and use the technical freedom offered by microservices.

Even if decisions are part of the micro architecture, a team can still standardize because using technologies with which other teams already had positive experiences decreases the risk and allows for synergy.

Macro architectural decisions

The **macro architecture**, on the other hand, contains those parts of the architecture that **apply uniformly to all microservices**.

The division into these two levels gives the individual microservices freedom, while simultaneously ensuring the integrity of the overall system.

In all cases, decisions must be made explicitly. The teams must consciously deal with the macro architecture and the freedoms in micro architecture. Micro and macro architecture form a trade-off which is different in each project.

In the next lesson, let’s take a quick quiz to go over what you have learned in this chapter!

Introduction

In this lesson, we'll get a walkthrough of what this chapter holds for us.

WE'LL COVER THE FOLLOWING

- Chapter walkthrough

Migration **from a deployment monolith to a microservices architecture** is the common case for introducing microservices. Most projects start with a deployment monolith that the team wants to split into microservices later because the deployment monolith has too many disadvantages.

Of course, it is also possible to implement a new system directly with microservices from scratch.

Chapter walkthrough

This chapter **provides an overview of the challenges involved** in migrating to a microservices system.

- The chapter discusses possible **reasons for migration**. In this way, readers can assess whether a migration makes sense in their context. The approach to migration depends on your objectives. Therefore, knowing possible reasons is helpful for choosing a migration strategy.
- The chapter then shows a **typical strategy for migration**, with some alternatives. In this way, readers can choose a migration approach suitable for their scenario.

QUIZ

1

Converting a monolithic system to a microservices system is called a _____.

COMPLETED 0%

1 of 2

In the next lesson, we'll discuss the possible reasons for migration!

Reasons for Migrating

In this lesson, we'll discuss a few common reasons for migrating to a microservices architecture.

WE'LL COVER THE FOLLOWING

- Microservices offer a fresh start
- The reasons are already known
- A typical reason: speed of development

When migrating to microservices, it is **important to know the objectives for taking this step**. Depending on the reasons that led to this decision, the procedure for implementing them may vary.

Microservices offer a fresh start

Especially when replacing legacy systems, microservices have several advantages. Let’s discuss them.

The code of the legacy system no longer needs to be used in the new microservices because the microservices are implemented separately from the legacy system. The **microservices offer an unencumbered restart**.

The code of a legacy system is often no longer maintainable, and the technologies are frequently outdated. Therefore, reusing the old code would hinder the development of a clean new system.

Microservices thus solve the most important challenges when dealing with legacy systems, because otherwise, a restart is difficult.

Migration to microservices has the potential to solve the problem with the legacy system. After migration to microservices, further migrations can be limited to one or a few microservices. A migration of the entire system will probably not be necessary again.

A typical **reason for a system migration is outdated technology**. In a microservices system, such a migration can take place step by step – that is, microservice by microservice.

Another migration reason is an **unmaintainable system**. In this case, each microservice can be replaced individually.

The reasons are already known

The reasons for a migration are in the end identical to the reasons for using microservices. These have already been discussed in detail in the last chapter and may include:

- Increased security
- Robustness
- Independent scaling of individual microservices

A typical reason: speed of development

A typical reason for introducing microservices is the lack of speed in developing with a deployment monolith. When many developers are working on a deployment monolith, they need to **closely coordinate their work**. This costs time and therefore **slows down development**.

But even with a small team, a deployment monolith can be problematic because the **deployment is quite huge**. The size makes continuous delivery difficult to implement, and each release requires a lot of testing.

QUIZ

1 A team of two, working on a popular monolithic website with a very small codebase that started 15 years ago want to migrate to a microservices system. What would the primary reason for the migration be?

In the next lesson, we'll study typical migration strategies.

Typical Migration Strategies

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

WE'LL COVER THE FOLLOWING

- A typical scenario
- Give a preference to asynchronous communication
- Give preference to UI integration
- Avoid synchronous communication
- Reuse old interfaces?
- Integrating authentication
- Replicating data
  - Replication should be done in one direction
- Black box migration
  - Choosing the first microservice for the migration
- Extreme migration strategy: all changes in microservices
- Further procedure: step-by-step migration

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.

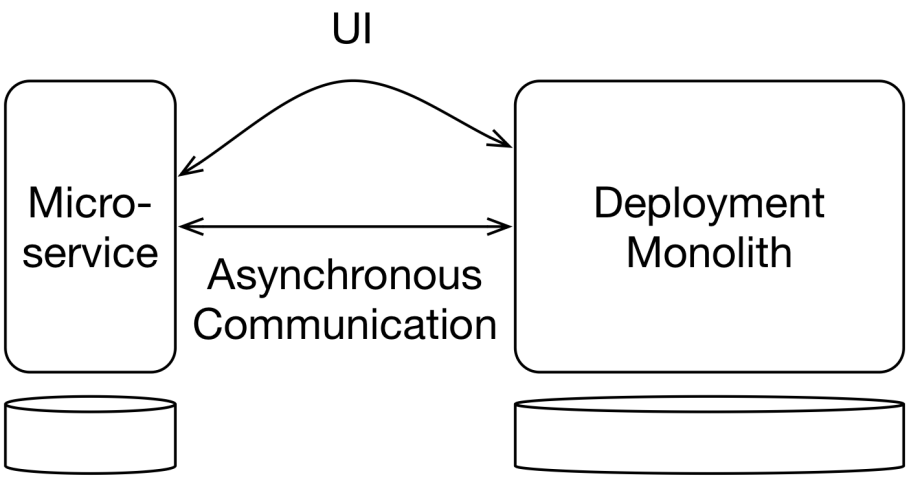
Because the migration process depends on the goals and on the structure of the legacy system, there is no universal approach. Therefore, the strategy presented here must not simply be used as is but must be adapted to the respective situation.

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.

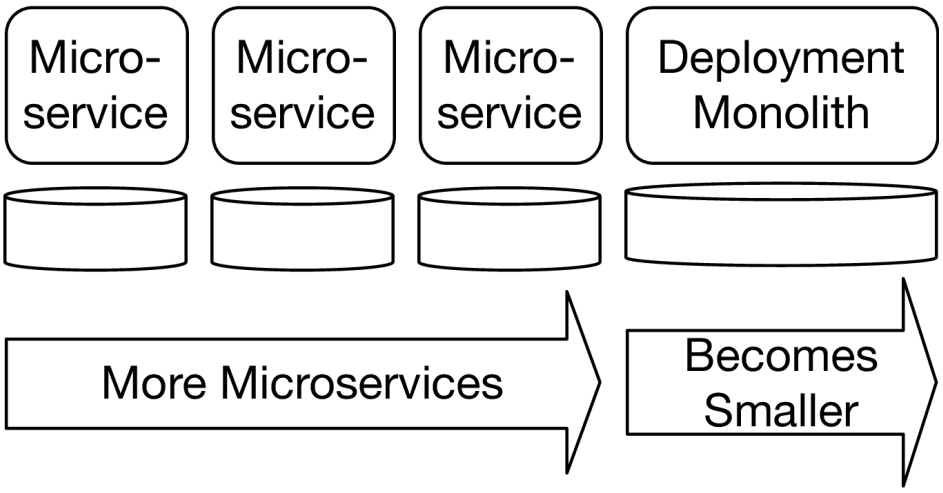
This automatically results in a migration to microservices as more and more logic is implemented in microservices over time. It is very easy to follow this rule.

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.

Alternative Strategies

In this lesson, we'll study a few alternative migration strategies.

WE'LL COVER THE FOLLOWING

- Goal: reliability
- Migration based on layers
- Copy/change
  - Caveats

There are many more migration strategies. There is a presentation that gives a good overview of the different approaches to the migration to microservices. Let's go over some of the most common approaches.

Goal: reliability

As mentioned previously, there can be very divergent approaches for the migration to microservices. The strategy depends mainly on the objectives to be achieved.

When the main objective for switching to microservices is **an increase in robustness**, at first, **reliability can be improved at the interfaces to external systems** or databases with libraries like Hystrix and Resilience4j.

Then, **the system can be split step by step into individual microservices that run independently** of each other so that a failure of one microservice no longer affects the other microservices. There is an interesting talk about this approach to which slides are available.

Migration based on layers

Another alternative is a **migration based on layers**.

For example, the UI can be migrated first. This can make sense when changes

For example, the UI can be migrated first. This can make sense when changes to the UI are imminent, and therefore the migration can be combined with

necessary changes.

Of course, this migration strategy is in contrast to the idea of combining UI, logic, and data in one self-contained system.

However, it can still be the first step towards this goal. In that case, **the remaining layers would have to be migrated into the same microservice afterward**. Alternatively, one stays with a division of microservices in layers, although it is not optimal. An ideal architecture, however, into which it is impossible to migrate, is a lot less helpful than a less optimal architecture that can actually be implemented.

Copy/change

Another possibility is copy/change. Here, the code of the **legacy system is copied**.

- In one copy, **one part of the system is developed further, while the other part is removed**.
- In a second copy, **it is the other way around**.

In this manner, the legacy system is converted into **two microservices**.

This approach has the **advantage** that the **old code is still used**, and therefore the functionalities of the microservices very likely correspond accurately to the functionalities of the legacy system.

Caveats

- However, **at the same time it is a great disadvantage to continue using the old code**. In most cases, the code of a legacy system is hard to maintain, and it is problematic to keep using this code.

- In addition, the database schema remains unaltered. The **shared use of the database schema by the legacy system and the microservices results in a tight coupling between the two**, which really should be avoided to be able to profit from the advantages microservices offer. That is why a black box migration might be better.

- Moreover, the structure of the legacy system and the technology stack largely remain the same. Thus, the project has a lot of **technical debt from the very beginning** and does **not represent a new start**.
- This approach does not take advantage of the benefits of microservices such as freedom of technology.

Therefore, **it should only be used in exceptional cases**.

QUIZ

1 When should the copy/change strategy be used?

COMPLETED 0%

1 of 2

In the next lesson, we'll study the build, operation, and organization concerned with turning a legacy system into a microservices system.

Build, Operation, and Organization

In this lesson, we'll discuss the build, operation, and organization pertaining to moving a legacy system to a microservices architecture.

WE'LL COVER THE FOLLOWING

- Co-existence between microservices and legacy systems
- Integration test of microservices and legacy systems
- Coordinated deployment between legacy systems and microservices
- Organizational aspects
- Recommendation: do not implement all aspects at once

Code migration alone is not enough to turn a legacy system into a microservices system.

- **The microservices must also be built**. A suitable tool must be selected for this purpose. In addition, the continuous integration server has to cope with the multitude of microservices.
- Similarly, technologies and approaches must be introduced to **enable the deployment and operation** of microservices.
- Finally, **a suitable test strategy must be established**. This also requires the automated setup of test environments and the assurance that the tests are independent.
    - For example, stubs that simulate microservices or the legacy system are useful for this purpose, as are consumer-driven contract tests. They safeguard the requirements for the interfaces of microservices or legacy systems with the help of tests.
    - However, legacy systems are often very complicated, so these techniques are difficult to implement.

Therefore, dealing with the first microservice can require extra effort because the infrastructure for build and deployment needs to be set up. It is conceivable to build the infrastructure later, but it is recommended to start building the infrastructure as early as possible in order to reduce the risk of migration.

One or a few microservices can still be operated with an inadequate solution for build and deployment. However, once the number of microservices increases, without appropriate infrastructure, the necessary effort will become so high that it can lead to project failure.

Co-existence between microservices and legacy systems

During migration, the legacy system must be deployed and further developed in addition to the microservices. It is unrealistic to deploy the legacy system as often as the microservices because the effort of deploying the legacy system is usually far too high.

Therefore, **changes affecting both the legacy system and the microservices are difficult to implement**. They require at least one deployment of the microservices and one deployment of the legacy system. Solutions can be found at the architectural level.

If a new feature is implemented only in a microservice, then the deployment of only this microservice is necessary. This speaks for a division of the microservices according to bounded context.

Another option would be to **integrate the monolith with patterns** such as *open host service* or *published language*, as described in the previous chapter, to provide a generic interface that rarely needs to be changed.

Integration test of microservices and legacy systems

There must also be integration tests that test microservices with the version of the legacy system currently in production and with the one currently being developed. This ensures that the microservices continue to work when the legacy system is deployed.

The legacy system can support two different versions of the interfaces, so that microservices can switch to a new version of an interface when it is provided. However, no microservice is forced to use a new interface that has not yet been tested together with the microservice. In this way, the version of the microservice that uses a new interface of the legacy system can be deployed at any time.

Coordinated deployment between legacy systems and microservices

Coordinated deployment of microservices together with the legacy system is an alternative. When a change is made, the new version of **the microservices and the legacy system are rolled out at the same time**. However, this approach has a few disadvantages:

- This increases the risk because more changes occur at the same time
- It is harder to roll back the deployment.
- It is also difficult to implement this approach without downtime.
- With a complex microservices environment, this option is hardly possible anymore because too many microservices would have to be deployed at once.

Therefore, the deployment of microservices and legacy systems should be decoupled from the outset.

Organizational aspects

An essential advantage of microservices is the possibility to scale the development process.

If the goal of migration to microservices is to have independent teams, the migration of the architecture must be accompanied by a reorganization. This lesson in the previous chapter discusses the essential aspects of the target organization.

The organizational change must be coordinated with the technical migration. For example, a microservice can be detached from the legacy system and then

developed autonomously by a team. At the same time, the other

organizational structures can be set up – for example, the ones required for defining the macro architecture.

Recommendation: do not implement all aspects at once

Microservices require changes in architecture and organization as well as the introduction of new technologies.

Implementing all these changes at once is risky and complicated. Unfortunately, many of the changes are connected. Without new technologies, the architecture is difficult to implement. Without the architecture, organizational changes are difficult to introduce.

However, making all these changes at once should still be avoided. For each change, the question as to whether the change is actually necessary should be asked, in order to implement it at a later point in time.

QUIZ

1 Why are changes that affect both the legacy system and the microservices difficult to implement?

You can pick more than one answer.

In the next lesson, we'll discuss a few variations on the approaches discussed in this chapter.

Variations

The ideas for migration can easily combine with many other approaches. Let's study a few.

WE'LL COVER THE FOLLOWING

- Combining approaches to migration
- Experiments

Combining approaches to migration

- The ideas concerning typical migration strategies from the lesson on typical migration strategies fit very well to the concept of **self-contained systems (SCS)**. The migration can therefore simply separate a part of the legacy system into an SCS.
- Rules for authentication or communication between microservices as well as between microservices and the legacy system can be the starting point of a **macro architecture** (see chapter 3). A domain macro architecture is very useful, which can also include the legacy system in addition to microservices.
- **Frontend integration** can make sense for the integration between the legacy system and microservices.
- **Asynchronous microservices** fit very well with migration because they allow for a loose coupling. Especially for a migration, it can be sensible to continue to use an existing messaging technology for asynchronous communication to minimize the effort.
- **Synchronous microservices** should be used cautiously because this creates a tight coupling and resilience becomes difficult.
- **Kubernetes, PaaS, or Docker** are also interesting in a migration scenario. However, they represent a *new environment* that needs to be

operated. It may, therefore, make sense to use a classical deployment and operation environment at least at the beginning to reduce the initial migration effort. In the long term, however, such environments have many advantages. In addition, of course, the old system can be operated in such an environment.

Experiments

The migration strategy must match the respective scenario. The following questions are important in order to design your own strategy.

- What are the **goals of the migration** to microservices?
  - Which are especially important?
  - What impact does this have on the migration strategy?

In principle, **migration should take place gradually**. The selection of the parts to be migrated to microservices can be made according to technical or domain criteria. However, domain criteria are better suited, at least in the long term.

The **following approach is suitable for a migration based on domain criteria**:

- Split the system into bounded contexts.
- Which of the bounded contexts will you migrate first? Why? Reasons can be the simple migration of the bounded context or many planned changes in the bounded context. Consider the different scenarios.

In the next lesson, we'll look at a quick chapter conclusion.

Chapter Conclusion

In this lesson, we'll formally conclude this chapter with some notes regarding migration.

WE'LL COVER THE FOLLOWING

- Summary

Summary

- Migration to microservices is the typical approach for the introduction of microservices. Implementing a completely new system with microservices is the exception, but of course, it is also possible.
- One of the most important advantages of microservices is that they **work well for scenarios other than greenfield projects**.
- Choosing the **right migration strategy** is a complex task. It **depends on the legacy system and migration goals**. This chapter shows a starting point from which each project must develop its own strategy.
- Because of the benefits of migration, microservices should be considered in any project meant to modernize a legacy system. Microservices enable step-by-step modernization, in which completely different technologies can be used. This is very helpful.
- The **migration strategy can have a significant impact** on the architecture and technology selection. A split into microservices similar to the split of the modules in the legacy system can greatly simplify migration. Such a compromise has far-reaching consequences and can lead to a worse target architecture, but it can still make sense.
- Finally, it must be possible to actually implement the architecture, and for this, a simple approach for the migration into the architecture is essential.

In the next chapter, we'll discuss Docker!

Introduction

In this lesson, we'll walk through what we will learn in this chapter.

WE'LL COVER THE FOLLOWING

- Chapter walkthrough
- Licences and projects

Chapter walkthrough

This chapter introduces Docker and covers the following:

- After studying the chapter, the reader is able to **run the examples provided** in the following chapters **in a Docker environment**.
- Docker and microservices are nearly synonymous. This chapter **explains why Docker fits so well with microservices**.
- Docker facilitates software installation. Important for this is the `Dockerfile`, which describes the installation of the software in a simple way.
- Docker Machine and Docker Compose support Docker on server systems and complex software environments with Docker.
- The chapter lays the foundation for an understanding of technologies such as **Kubernetes Cloud Foundry**, which are based on Docker.

Licences and projects

Docker is under the Apache 2.0 license. It is developed by the company Docker, Inc., among others.

Some core components, such as Moby, for example, are under an Open Source license and allow other developers to implement systems similar to Docker.

Docker is based on Linux containers, which isolate processes in Linux systems from each other. The Open Container Initiative ensures via standardization of the compatibility of the different container systems.

QUIZ

1

A `Dockerfile` describes the installation of software in a simple way.

COMPLETED 0%

1 of 2

In the next lesson, we'll discuss some reasons for using Docker for microservices.

Docker for Microservices: Reasons

In this lesson, we'll look at some reasons for using Docker for microservices.

WE'LL COVER THE FOLLOWING

- OS processes for microservices
- Virtual machines: too heavy-weight for microservices
  - Overhead

Chapter 2 defined microservices as separately deployable units. The separate deployment not only results in a decoupling at the architectural level, but also in regard to technology choice, robustness, security, and scalability.

OS processes for microservices

If microservices are supposed to have all these characteristics, the question arises as to **how they can be implemented**. Microservices must be scalable independently of each other. In the event of a crash, a microservice must not be allowed to make other microservices unavailable, too, and thus endanger the robustness of the whole system. Therefore, **microservices must at least be separate processes**.

Scalability can be guaranteed by multiple instances of a process. When an application is started, the operating system generates a process and allocates resources such as CPU or memory to it. Therefore, more processes can use more resources.

But processes are limited concerning scaling. If the processes all run on one server, then only a limited amount of hardware resources are available. Instead, the microservices should run in a cluster. Kubernetes and Cloud Foundry support running microservices in a cluster.

With processes, robustness is guaranteed to a certain extent because the

crash of one process does not affect the other processes. However, a server failure still causes a large number of processes, and thus microservices, to fail.

But there are also other problems:

- **All processes share one operating system**. It must provide the libraries and tools for all microservices. Each microservice must be compatible with the operating system version. It is difficult to configure the operating system to support all microservices.
- In addition, the processes must coordinate in such a way that each process **has its own network port**. If you have a large number of processes, it becomes increasingly harder to find unused ports. Also, it's hard to figure out which ports are used by which process.

Virtual machines: too heavy-weight for microservices

Instead of a process, **each microservice can run in its own virtual** machine.

> **Virtual machines** are simulated computers that run on the same physical hardware. For the operating system and application, virtual machines look exactly like a physical server.

Through virtualization, the microservice has its own operating system installation. Thus, the **configuration of the operating system can be adapted to the specific microservice**, and there is also **complete freedom in choosing the network port**.

Overhead

However, a virtual machine has a **substantial overhead**:

- The virtual machine must give the operating system the illusion of running directly on the hardware. This leads to overhead. Therefore, **performance is poorer** than with physical hardware.
- Each microservice has its own instance of the operating system. This **consumes a lot of memory** in the *RAM*.
- Finally, the virtual machine has virtual disks with a complete operating

system installation. This means that the microservice **occupies a lot of hard disk space**.

So virtual machines have an overhead, making their operation expensive. In addition, operations will have to **manage a large number of virtual servers**. This is **complicated and time-consuming**.

The **ideal solution** would be a **lightweight alternative to virtualization**, which possesses the isolation of virtual machines, but consumes as little resources as processes do and is similarly easy to operate.

QUIZ

1 Which of the following is NOT a reason why microservices must at least be implemented as separate processes?

COMPLETED 0%

1 of 4

In the next lesson, we'll discuss some Docker basics.

Docker Basics

In this lesson, we'll go over some Docker basics.

WE'LL COVER THE FOLLOWING

- What is Docker?
  - Shared kernel
  - Isolated network of Dockers
  - Optimized file system
- One process per container
- Docker image and Docker registry
- Supported operating systems
- Operating systems for Docker
  - One process per container
  - Host does not have to have tools needed in the containers
- Overview
- Does it always have to be docker?
- Microservices as WARs in Java application servers
  - Disadvantages
  - Advantages

What is Docker?

Docker represents a lightweight alternative to virtualization. Although Docker does not provide as much isolation as a virtualization, it is practically as lightweight as a process.

Shared kernel

Instead of having a complete virtual machine of their own, Docker containers *share* the *kernel* of the operating system on the Docker host. The Docker host is

the system on which the Docker containers run. The processes from the

containers, therefore, appear in the process table of the operating system on which the Docker containers are running.

Isolated network of Dockers

The Docker containers have their **own network interface**. In this way, the **same port can be used in each Docker container**, and each container can use any number of ports.

The network interface is in a subnet where all Docker containers are accessible. The subnet is not accessible from the outside. This is at least the standard configuration of Docker.

The Docker network configuration offers many other alternatives. To still allow external access to a Docker container from the outside, **ports of a Docker container can be mapped to ports on the Docker host**. When mapping the ports of the Docker containers to the ports of the Docker host, be careful because each port of the Docker host can only be mapped to one port of a Docker container.

Optimized file system

Finally, the file system is optimized. There are **layers in the file system**. When a microservice reads a file, it goes through the layers from top to bottom until it finds the data. The containers can share layers.

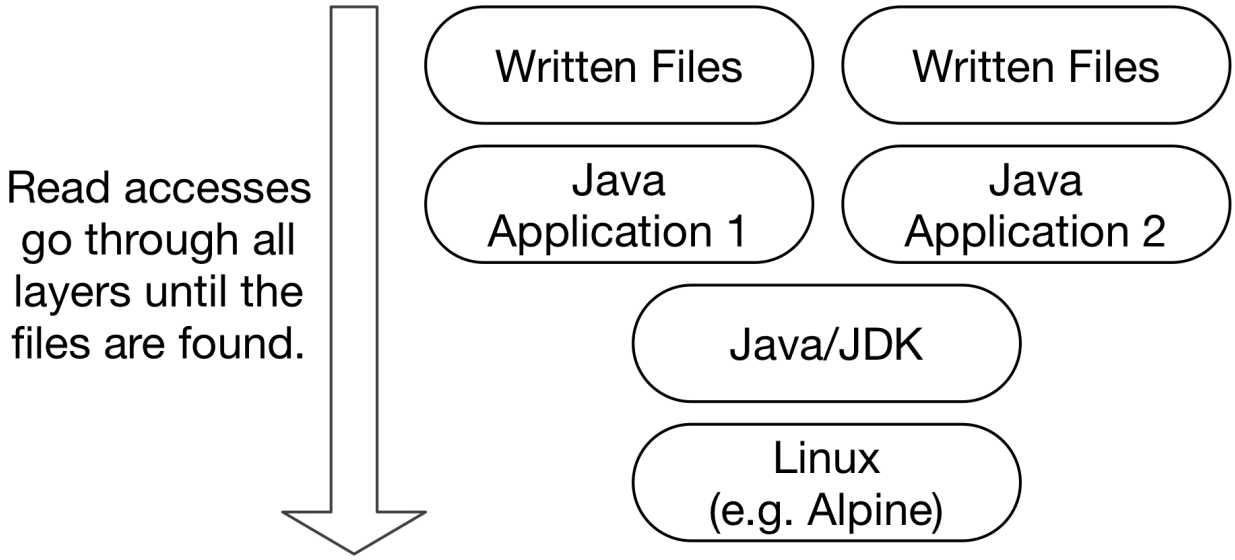
The drawing below shows this more precisely. The file system layer at the bottom represents a simple Linux installation with the Alpine Linux distribution. Another layer is the Java installation. Both applications share these two layers, which are stored only once on the hard disk, although both microservices use them.

Only the applications are stored in file system layers that are exclusively available to a single container. The lower layers cannot be changed. The microservices can only write to the top layer. The reuse of the layers reduces the storage requirements of the Docker containers.

It is easily **possible to start hundreds of containers on a laptop**. This is not surprising: after all, it is also possible to start hundreds of processes on a

laptop. Docker has **no significant overhead** compared to a process.

Compared to virtual machines, however, the **performance benefits are outstanding**.



File System Layers in Docker

One process per container

Ultimately, Docker containers are highly isolated processes due to their own network interface and file system.

Therefore, **only one process should run in a Docker container**. Running more than one process in a Docker container contradicts the idea of separating processes by means of Docker containers. Because only one process is supposed to run in a Docker container, there should be no background services or daemons in Docker containers.

Docker image and Docker registry

File systems of Docker containers can be exported as Docker images. These images can be passed on as files or stored in a Docker registry.

Many repositories such as Nexus and Artifactory can store and provide Docker images just like compiled software and libraries. This makes it easy to exchange Docker images with a Docker registry for installation in production. The transfer of images from and to the registry is optimized. Only the updated layers are transferred.

Supported operating systems

Supported operating systems

Docker **was originally a Linux technology**. For operating systems such as **macOS and Windows, Docker installations are available**. For this purpose, a virtual machine with a Linux installation is running in the background. This is transparent for the user. It seems as if the Docker containers are running directly on a computer.

In addition, Windows, since Windows Server 2016, provides Windows Docker containers. Linux applications run in a Linux Docker container and Windows applications in a Windows Docker container.

Operating systems for Docker

Docker changes the requirements for operating systems.

One process per container

Only one process is supposed to run in one *Docker container*. This means that only as much of the operating system is required as is needed to run that process.

- For a Java application, we use the Java Virtual Machine (JVM), which requires some Linux libraries that are loaded at runtime. A shell is not necessary, for example. Therefore, distributions like Alpine Linux, which are just a few megabytes in size, only contain the most important tools, making them an ideal basis for Docker containers.
- The programming language, Go, can create statically linked programs. In that case, nothing else has to be available in the Docker container besides the program itself, and no Linux distribution is required at all.

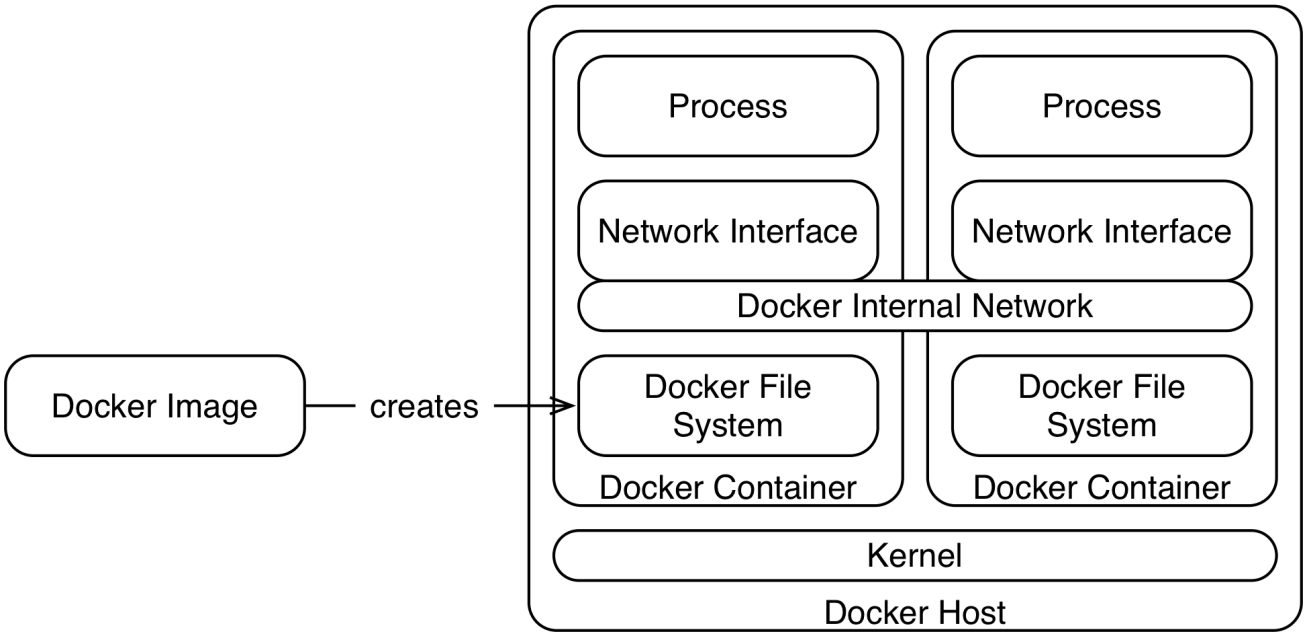
Host does not have to have tools needed in the containers

The *Docker host* on which the Docker containers run **only has to run Docker containers**. Many Linux tools are therefore superfluous.

- CoreOS is a Linux distribution that can run little more than Docker containers and, for example, considerably simplifies operating system updates of an entire cluster.
- CoreOS can also serve as a basis for Kubernetes.

Overview

The drawing below shows an overview of the Docker concepts.



Overview of Docker

- The **Docker host** is the machine on which the Docker containers run. It can be a *virtual* machine or a *physical* machine.
- **Docker containers** run on the Docker host.
- The containers typically contain a **process**.
- Each container has its own **network interface** with its own IP address. This network interface is only accessible from the Docker internal network. However, there are also ways to allow access from outside this network.
- In addition, each container has its own **file system**.
- When a container is started, the **Docker image** creates the first version of the Docker file system. When the container has been started, the image is extended by another layer into which the container can write its own data.
- All Docker containers share the **kernel** of the Docker host.

Does it always have to be docker?

Does it always have to be docker?

Docker is a very popular option for deploying microservices. However, there are **alternatives**. Two alternatives were already mentioned in this lesson: **virtual machines** or **processes**.

Microservices as WARs in Java application servers

However, you can also deploy several microservices as **WAR files** in a Java application server or Java web server. WARs contain a Java web application, and can be deployed separately.

However, the deployment of a WAR may require the server to be restarted. Because microservices should only be deployable separately as per chapter 2, **microservices can be implemented as WARs**.

Disadvantages

- **Compromises are made with regards to robustness**. A memory leak in a microservice can lead to failure of all microservices. The microservice would allocate more and more memory until an `OutOfMemoryError` is thrown and the entire Java application server crashes.
- **Separate scalability is also difficult to implement** because each server contains all microservices and, therefore, only all microservices can be scaled together.
    - This makes the scaling more complex than in cases where each server contains only one microservice as unneeded microservices are also scaled.
    - Of course, it would be possible to run each WAR on a separate Java web server and have multiple instances of each of these Java web servers. But then the WARs would no longer run together on one Java web server.
- And finally, all microservices run in one operating system process, which is a **compromise in terms of security**. When a hacker can take over the process, he or she has access to the entire functionality and data of all microservices.

Advantages

- In return, these approaches **consume less resources**. An application server with several web applications requires only one JVM (Java Virtual Machine), only one process, and only one operating system instance.
- Furthermore, there is **no need to introduce a new infrastructure** if application servers are already in use, which can reduce the workload for operations.

QUIZ

1

Consider an application where all the microservices have to be individually externally accessible from the network. Does Docker solve the issue of keeping track of unused ports that the host machine has?

COMPLETED 0%

1 of 3

In the next lesson, we'll study `Dockerfiles`!

Dockerfiles

In this lesson, we'll discuss Dockerfiles.

WE'LL COVER THE FOLLOWING

- Overview
- An example for a Dockerfile
- File system layers in the example
- Problem with caching and layers
- Docker multi-stage builds
- Immutable server with Docker
- Docker and tools: Puppet, Chef or Ansible

Overview

The creation of Docker images is done via files named `Dockerfile`. One of Docker's strengths is that Dockerfiles are easy to write and therefore, the rolling out of software can be automated without any problems.

The typical components of a `Dockerfile` are:

- `FROM` defines a base image on which the installation is based. A base image for a microservice usually contains a Linux distribution and basic software, such as the JVM, for example.
- `RUN` defines commands that execute to create the Docker image. In essence, a `Dockerfile` is a shell script that installs the software.
- `CMD` defines what happens when the Docker container is started. Typically, only one process should run in one Docker container. This is started by `CMD`.
- `COPY` copies files in the Docker image. `ADD` does the same; however, it can

also unpack archives and download files from a URL on the Internet. `COPY` is simpler to understand because it does not extract archives, for example. Also, from a security perspective, it can be problematic to download software from the Internet into Docker containers. Therefore, `COPY` should be given preference over `ADD`.

- `EXPOSE` exposes a port of the Docker container. This can then be contacted by other Docker containers or can be tied to a port of the Docker host.

A comprehensive reference is available on the Internet which contains additional details to the commands in `Dockerfile`.

An example for a Dockerfile

A simple example of a `Dockerfile` for a Java microservice looks like this:

```
FROM openjdk:11.0.2-jre-slim
COPY target/customer.jar .
CMD /usr/bin/java -Xmx400m -Xms400m -jar customer.jar
EXPOSE 8080
```

- The first line defines the base image with `FROM`. It is downloaded from the public Docker hub. The image contains a Linux distribution and a Java Virtual Machine (JVM).

- The second line adds a JAR file to the image with `COPY`. A JAR file (Java ARchive) contains all components of a Java application. It has to be available in a sub directory `target` below the directory in which the `Dockerfile` is stored. The JAR file is copied into the root directory of the container.

- The `CMD` entry determines which process should be started when the container is started. In this example, a Java process runs the JAR file.

- Finally, `EXPOSE` makes a port available to the outside. This is the port under which the application is available. `EXPOSE` only means that the container provides the port. It is then available on the internal Docker network. Access from outside is only possible when this is enabled at the start of the container.

The Docker image can be built with the command `docker build --tag=microservice-customer microservice-customer`.

`docker` is the command line tool with which most functionalities of Docker can be controlled. The created Docker image has the tag `microservices-customer` as defined by the `--tag` parameter.

The `Dockerfile` has to be in the sub directory `microservice-customer`. The name of this directory is the second parameter.

File system layers in the example

The first image in this lesson shows that a Docker image consists of multiple layers.

Although no layers have been defined in `Dockerfile`, the image `microservices-customer` contains multiple layers. Each line of the `Dockerfile` defines a new layer. These layers are reused. Thus, if `docker build` is called again, Docker will go through the `Dockerfile` again. However, it will find that all actions in `Dockerfile` have already been executed once. As a result, nothing happens.

If the `Dockerfile` was modified in such a way that, after the `COPY`, another line with a `COPY` of another file is inserted, Docker would reuse the existing layer with the first `COPY`, but the second `COPY` and all further lines would then create new layers.

In this manner, Docker only re-creates the layers that need to be rebuilt. This not only saves storage space but is also much faster.

Problem with caching and layers

A `Dockerfile` for obtaining a Ubuntu installation with updates looks like this:

```
FROM ubuntu:15.04
RUN apt-get update ; apt-get dist-upgrade -y -qq
```

First, a Ubuntu base image is loaded from the public Docker hub on the Internet. The commands `apt-get update` and `apt-get dist-upgrade -y -qq` are used to update the package index and then install all packages with updates. The options ensure that `apt-get` does not ask the user for permission and

outputs only a few messages on the console.

The two commands are separated in the line by a `;`. This causes them to be executed one after the other. A new file system layer is created only after both commands have been executed. This is useful for creating more compact images with fewer layers.

However, this `Dockerfile` also has a problem. If the Docker image is built again, no current updates will be downloaded. Instead, nothing happens because the images are already there.

Layer caching is based only on commands. Docker does not recognize that the external package index has changed. To ignore the existing images and force the rebuilding of the images, the parameter `--no-cache=true` can be passed to `docker build`.

Docker multi-stage builds

Everything needed to build a Docker image can also be found in the Docker image. Therefore, all of it is available at runtime of the Docker container.

If code is being compiled in a `Dockerfile`, the compiler is also available at runtime. This is unnecessary and can even be a security problem. If the container is compromised, the attacker can compile code inside the container with the original tools, which might allow more attacks.

It is not easy to delete all of the build environment because today there is usually a complex tool chain for building software.

Building the software outside of Docker might also not be an option. Docker is based on Linux; therefore, on macOS, you would need to run a cross compiler to generate Linux binaries.

To solve this problem, Docker has **Multi Stage Builds**. They make it possible to compile the program in one phase of the build in a Docker image, and to transfer only the compiled program to the next phase into a different Docker image.

Afterwards, the build tools are no longer available at runtime. They do not have to be deleted, and they do not have to be installed on the host machine.

This lesson in the next chapter shows a Docker Multi Stage Build using a Go program as example.

Immutable server with Docker

Immutable server is an idea that predates Docker. The idea of an immutable server is that **a server will never be changed**; therefore, the software on the server will never be updated or modified. The server will always be completely rebuilt from scratch.

In this way, the state of the server can be reconstructed cleanly. For each server, an installation script installs all the needed software on a basic OS image.

However, **immutable servers are hard to implement**. It is very cumbersome to completely reinstall a server. The process might take minutes or hours. That is far too much time compared to, for example, just changing a configuration file.

This is exactly where **Docker helps**. Because of the optimizations, only the necessary steps are taken so that *immutable servers* can also be an option from this perspective.

A `Dockerfile` describes how to create a Docker image starting from a base image. With each build, it will seem as if the complete Docker image is being created. Behind the scenes, however, optimizations ensure that only what is really necessary is built.

For example, if in the very last step a configuration file is added and just that configuration file has been modified, Docker is smart enough to reuse the results of all other installation steps and just add the new configuration file. This just takes a few seconds.

Docker is conceptually as clear as an immutable server but much more efficient for actually implementing them.

Docker and tools: Puppet, Chef or Ansible

Besides immutable servers, there are other ways to handle the installation of software.

Idempotent installation means that an installation script provides the same results no matter how often it runs. For an idempotent installation, there are no steps like “install the Java package,” but rather a definition of the desired state: “ensure that the Java package is installed”. If the installation is run on a fresh OS, Java would be installed. If the installation is run on a system that already has Java installed, nothing happens.

Idempotent installation is particularly **useful to enable updates**. For each update, the script would check if all software is installed in the correct version. If that is not the case, the correct version is installed. Tools such as **Puppet, Chef, or Ansible** support the concept of idempotent installation.

A `Dockerfile` is a very easy way to install software. The use of tools such as Puppet, Chef, or Ansible for the installation of software in a Docker image is possible but does not make a lot of sense. In particular, it is not necessary to use the update functionalities of these tools, because the image is typically freshly built with the *immutable server* approach.

This approach is easier than writing Puppet, Chef or Ansible scripts because defining the desired state is usually quite complex. The `Dockerfile` only describes how to build an image, whereas the other tools must also enable updates of the servers and are therefore more difficult to use.

QUIZ

1 Which of the following best describes how the `COPY` command works?

In the next lesson, we'll study Docker Compose.

Docker Compose

In this lesson, we'll discuss Docker Compose.

WE'LL COVER THE FOLLOWING

- Overview
- Service discovery with Docker Compose links
  - Ports
  - Volumes
- YAML configuration
  - Additional options
- Docker Compose live environment!
- Docker Compose commands

Overview

A typical microservice system contains more than a single Docker container. As explained in chapter 2, microservices are modules of a system.

It would be good to have a way to start and run several containers together for starting all the modules that the system consists of in one go. This can be done with Docker Compose.

Service discovery with Docker Compose links

Coordinating a system of multiple Docker containers requires more than just starting multiple Docker containers. It also requires **configurations for the virtual network** with which the Docker containers communicate with each other. In particular, **containers must be able to find each other in order to communicate**.

In a Docker Compose environment, **a service can simply contact another**

service via a Docker Compose link and then use the service name as the

hostname. So it could use a URL like `http://order/` to contact the order microservice.

Docker Compose links offer some kind of service discovery, that is, a way for microservices to find other microservices. Synchronous microservices require a form of service discovery.

Docker Compose links extend Docker links. Docker links only allow communication. Docker Compose links also implement **load balancing** and set the start order so that the dependent Docker containers start first.

Ports

In addition, Docker Compose can bind ports from the containers to the ports of the Docker host where the Docker containers run.

Volumes

Docker Compose can also provide volumes. These are file systems that can be shared by multiple containers. This allows containers to communicate by exchanging files.

YAML configuration

Docker Compose configures the interaction of the Docker containers with a YAML configuration file `docker-compose.yml`.

The following file comes from a project which implements Edge Side Includes as a way to compose websites from different sources. For this purpose, three containers must be coordinated.

- `common` is a web application that is supposed to deliver common artifacts.
- `order` is a web application for processing orders.
- `varnish` is a web cache to coordinate the two web applications.

```
version: '3'
services:
  common:
    build: ../scs-demo-esi-common/
```

```
order:
  build: ../scs-demo-esi-order
varnish:

  build: varnish
  links:
   - common
   - order
  ports:
   - "8080:8080"
```

- The first line defines the used version of Docker Compose – in this case three.
- The second line starts the definition of the services.
- Line three defines the service `common`. The directory specified in line four contains a `Dockerfile` with which the service can be built. An alternative to `build` would be `image` to use a Docker image from a Docker registry.
- The definition of the service `order` also specifies a directory with a `Dockerfile`. No other settings are required for this service (lines 5/6).
- The service `varnish` is also defined by a directory with a `Dockerfile` (lines 7/8).
- The service `varnish` must have Docker Compose links to the services `common` and `order`. Therefore, it has entries under `links`. The `varnish` service can therefore reach the other services using the host names `common` and `order` (lines 9-11).
- Finally, port 8080 of the service `varnish` is bound to port 8080 of the Docker host, on which Docker containers run (lines 12-13).

Additional options

Further elements of the YAML configuration are described in the reference documentation. For example, Docker Compose supports volumes shared by multiple Docker containers. Docker Compose can also configure the Docker containers using environment variables.

Docker Compose live environment!

You can try out Docker compose commands in the following environment. It consists of 3 Docker images running together. The final app should be

available at the generated link such as https://x6jr4kg.educative.run

Docker Compose commands

Docker Compose is controlled by the command line tool `docker-compose`. It must be started in the directory where the file `docker-compose.yml` is stored. The reference documentation lists all command line options for this tool. This lesson in the Appendix shows an overview of the Docker Compose commands. The most important ones are:

- `docker-compose build` builds the images for the services. **Try this command in the Docker compose coding environment above**!
- With `docker-compose up`, all services are started. The command returns the combined standard output of all services. This is rarely helpful, so `docker-compose up -d` is often the better choice. In this case, the standard output is not returned. **Try this command next**! Notice all the services will be up and accessible at the given link. With `docker log` the output of individual containers can be viewed.
- With `docker-compose up --scale <service>=<number>`, a larger number of containers for a service can be started. In the example from the listing, `docker-compose up --scale common=2` could ensure that two containers for the service `common` are started. **Try this next**.
- `docker-compose down` shuts down all services and deletes the containers. **Try this to shut down all services**.

Since the examples often require the interaction of several Docker containers, most examples have a `docker-compose.yml` file to run the containers together.

QUIZ

1 What is the purpose of Docker Compose?

In the next lesson, we'll discuss some variations to what we have already learned.

Variations

In this lesson, we'll discuss some variations to the docker approaches we've already discussed.

WE'LL COVER THE FOLLOWING

- Clusters
- Docker without Scheduler
- PaaS
- Experiments

There are not really any fundamental alternatives to Docker:

- **Virtualization** has too much overhead.
- **Processes** are not sufficiently isolated. The required libraries and runtime environments for all microservices must be installed in the operating system. This can be difficult because each process occupies one port, and therefore the allocation of ports must be coordinated.
- Other **container solutions** such as rkt are far less common.

Clusters

For production, **applications should run in a cluster**. Only in this way can the system be scaled across several servers and secured against the failure of individual servers.

Docker Compose can use Docker Swarm Mode for **cluster management**. Docker Swarm Mode is built into Docker.

Kubernetes is widely used for operating Docker containers in a cluster.

There are also other systems like **Mesos**. Mesos is actually a system for managing batches for data analysis in a cluster but it also supports Docker

containers. Offers in the cloud are also available, such as ECS (EC2 Container Service).

kubernetes

Docker without Scheduler

An alternative is to install Docker containers **directly on a server**. In this scenario, the servers are provided with classic mechanisms – for example, with virtualization. Linux or Windows is installed on the server. The microservice runs in a Docker container.

The only difference to the procedure without Docker is that Docker containers are now used for deployment. But this already makes it much easier to keep production and test environments identical. Concepts such as *immutable servers* are also easier to implement, as is the technology freedom for microservices.

Traditional virtualization is still responsible for high availability, scaling, and distribution to the servers.

This approach offers some of the advantages of Docker and reduces the effort.

These approaches have **one thing in common**: *how* the load is distributed in the cluster is decided by the scheduler – that is, by Kubernetes or Docker Swarm Mode. This means that the **scheduler is of crucial importance** for fail-safety and load balancing.

If a container fails, a new container must be started. Likewise, additional containers must be started at times of high loads, ideally on machines that are not too busy.

not too busy.

Schedulers such as Kubernetes solve many challenges, especially with synchronous microservices. It is highly recommended to use such a platform for operating microservices.

But the introduction of microservices requires many changes. The architecture needs to be adapted, and developers need to learn new approaches and technologies, as well as having adopted the deployment pipeline and the tests.

Implementing an additional technology such as a Docker scheduler should, therefore, be well thought out because many other changes also have to be made.

PaaS

Another alternative is a **PaaS**. A PaaS has a higher degree of abstraction than Docker schedulers because only the application needs to be provided.

The PaaS creates the Docker images. Therefore, a PaaS can be the simpler and therefore better solution.

Experiments

- Docker Machine can use clouds like Amazon Cloud or Microsoft Azure with the appropriate drivers. Create an account with one of the cloud providers. Most cloud providers offer free capacity to a new user.
- Create an account in the Docker Hub. Build a Docker image, for example, based on one of the microservices examples of the following chapters and place it in the Docker hub with `docker push`.
- Use the tutorials to familiarize yourself with the Docker Swarm Mode, which can be used to run Docker in a cluster.

QUIZ

1 Why should applications run in a cluster?

COMPLETED 0%

1 of 2

In the next lesson, we'll look at a quick conclusion of this chapter.

Chapter Conclusion

In this lesson, we'll formally conclude this chapter with a quick summary of what we've learned.

WE'LL COVER THE FOLLOWING

- Summary

Summary

Docker is a lightweight alternative for deploying and operating microservices. A microservice with all its dependencies can be packed into a Docker image and can then be well isolated from other microservices as a Docker container.

Virtual machines appear too heavyweight by comparison, whereas simple processes do not provide the necessary isolation.

Docker makes it easier to deploy the software. It is only necessary to distribute Docker images. `Dockerfiles` are used for this purpose, which are very easy to write. Concepts such as **immutable server** are also much easier to implement.

With **Docker Compose**, multiple containers can be coordinated to thereby build and launch an entire system of microservices in Docker containers.

Docker Machine can very easily install Docker environments on servers.

However, **Docker requires rethinking regarding operation**. Therefore, in some cases, alternatives might be helpful. This can be, for example, the deployment of several Java web applications on a single Java web server.

In the next chapter, we'll study technical micro architecture.

Introduction

In this lesson, we'll get a walkthrough of what this chapter holds for us.

WE'LL COVER THE FOLLOWING

- Why are microservices so important?
- Chapter walkthrough

Technical MicroArchitecture

Why are microservices so important?

> One of the **strengths** of microservices is that different technologies can be used in *each individual microservice*.

The technologies in the microservices can be defined as part of the microarchitecture (see chapter 3).

However, there are **technical challenges** to consider when **selecting technologies** for microservices.

Chapter walkthrough

This chapter explains **how to deal with the technical microarchitecture:**

- The reader gets to know the **requirements** regarding, e.g., operation or resilience, which the microarchitecture has to fulfill.
- Often microservices are implemented with **reactive technologies**. Thus, the chapter discusses this option in more detail and explains when this

approach makes sense.

- As a concrete example of technical microarchitecture, the chapter shows **Spring Boot** and **Spring Cloud**.
- Based on Spring Boot and Spring Cloud, the chapter shows how the **technical requirements the microarchitecture** has to address can be **fulfilled**.
- In addition, the chapter shows how the **programming language Go** in conjunction with appropriate frameworks fulfills the requirements **for implementing microservices.**

QUIZ

1 Which technology are microservices often implemented with as stated above?

COMPLETED 0%

1 of 2

In the next lesson, we'll start with the first point from the list above and discuss the requirements, a technology for implementing microservices has to fulfill.

Requirements

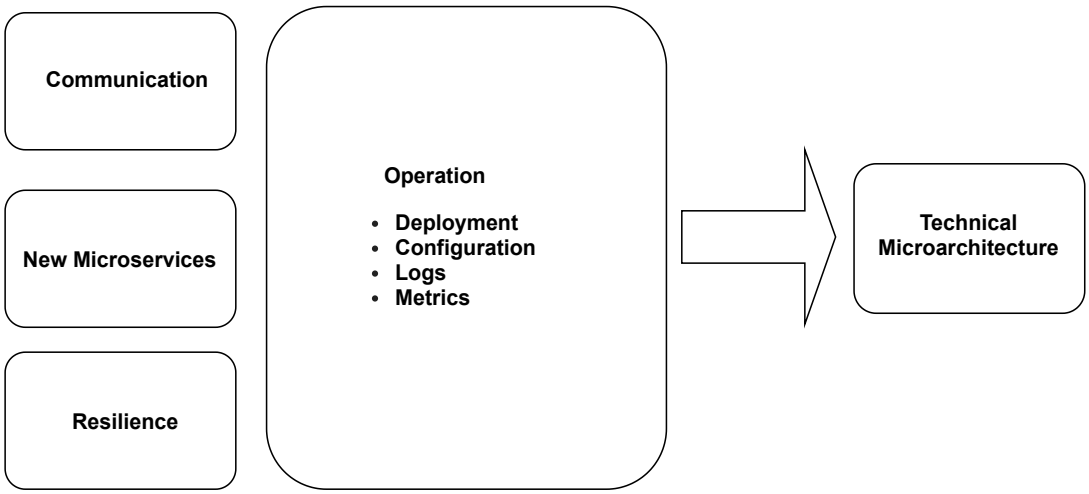
In this lesson, we'll discuss the requirements a technology for implementing microservices has to fulfill.

WE'LL COVER THE FOLLOWING

- Communication
- Operation
  - Configuration
  - Deployment
  - Logs
  - Metrics
- New microservices
  - Option 1: Microservices increase in size
  - Option 2: Constant size microservices increase in number
- Resilience

A technology for implementing microservices has to fulfill different requirements. The figure below gives us a birds-eye view of what these are.

We'll be discussing each of these in detail below.



Influencing Factors for the Technical Micro Architecture

Communication

Microservices have to **communicate** with *other microservices*. This requires **UI integration** in the **web UI** or **protocols** such as **REST** or **messaging**.

> It is a *macro architecture decision* which communication protocol is used (see Architecture Decisions).

However, the microservices have to support the chosen communication mechanism. Therefore,

> The macro architecture decision influences the micro architecture.

The technology choices at the micro architecture level have to ensure that the communication protocol defined by the macro architecture can really be implemented in each microservice.

In principle, every modern programming technology can support the typical communication protocols. Therefore, this requirement does not represent a real restriction.

Operation

Operating the microservices should be as easy as possible.

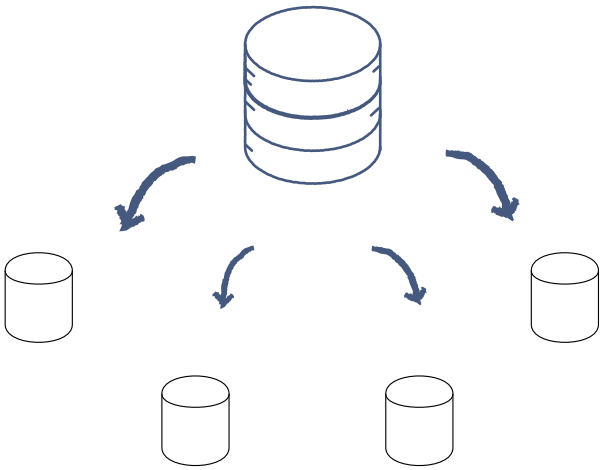
Topics in this area are:

- Deployment
- Configuration
- Logs
- Metrics

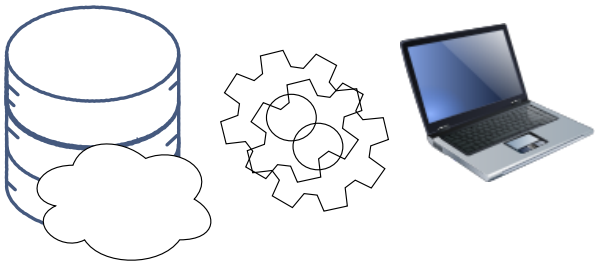
Let's cover these one by one:

Configuration

- The microservice has to be adapted to different scenarios. It is possible to use custom code for reading the configuration. However, an existing library can facilitate this task and promote a uniform application configuration.



Deployment



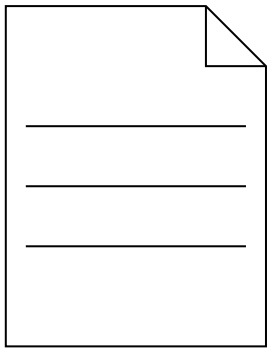
- The microservice has to be installed in an environment and has to run in this environment.

Logs

Writing log files is easy. However, the format should

be uniform for all microservices.

In addition, a simple log file is not enough when a server has to collect the logs from all microservices and provide them for analysis.



Therefore, technologies have to be in place for formatting the log outputs and for sending them to the server where all logs are stored and analyzed.

Metrics

Metrics have to be delivered to the central monitoring infrastructure.

This requires appropriate frameworks and libraries. In principle, different libraries can be used for implementing a macro architecture rule for which instance predefines a log format and a log server.

In this case, the micro architecture has to choose a library for the microservice. Macro architecture rules can also determine the library.

However, this limits the technological freedom of the microservices to those programming languages which can use the chosen library.

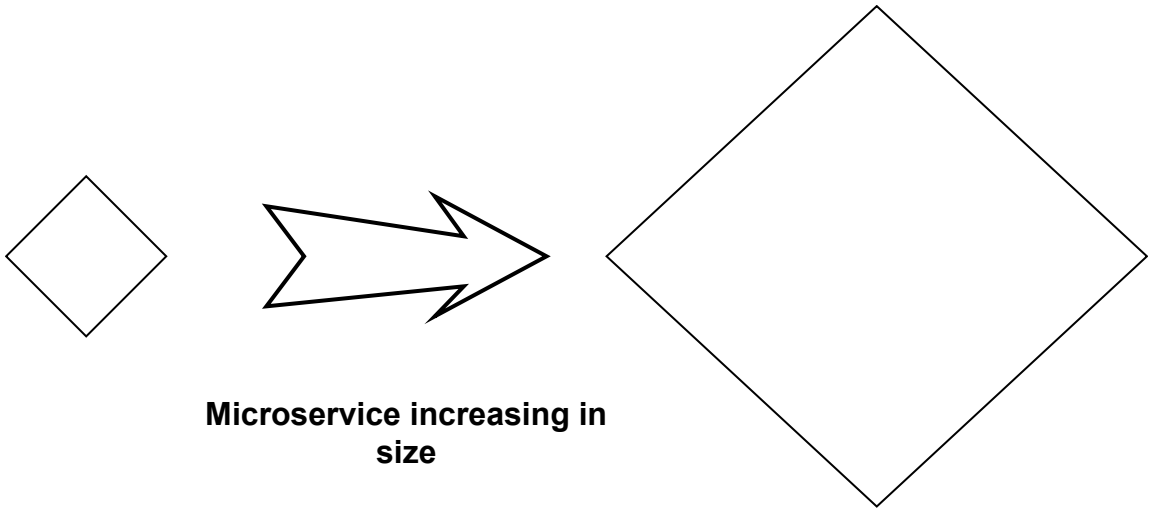
New microservices

It should be easy to create new microservices. When a project over time accumulates more and more code, there are two options:

1. The microservices become larger.
2. The number of microservices of constant size increases.

Option 1: Microservices increase in size

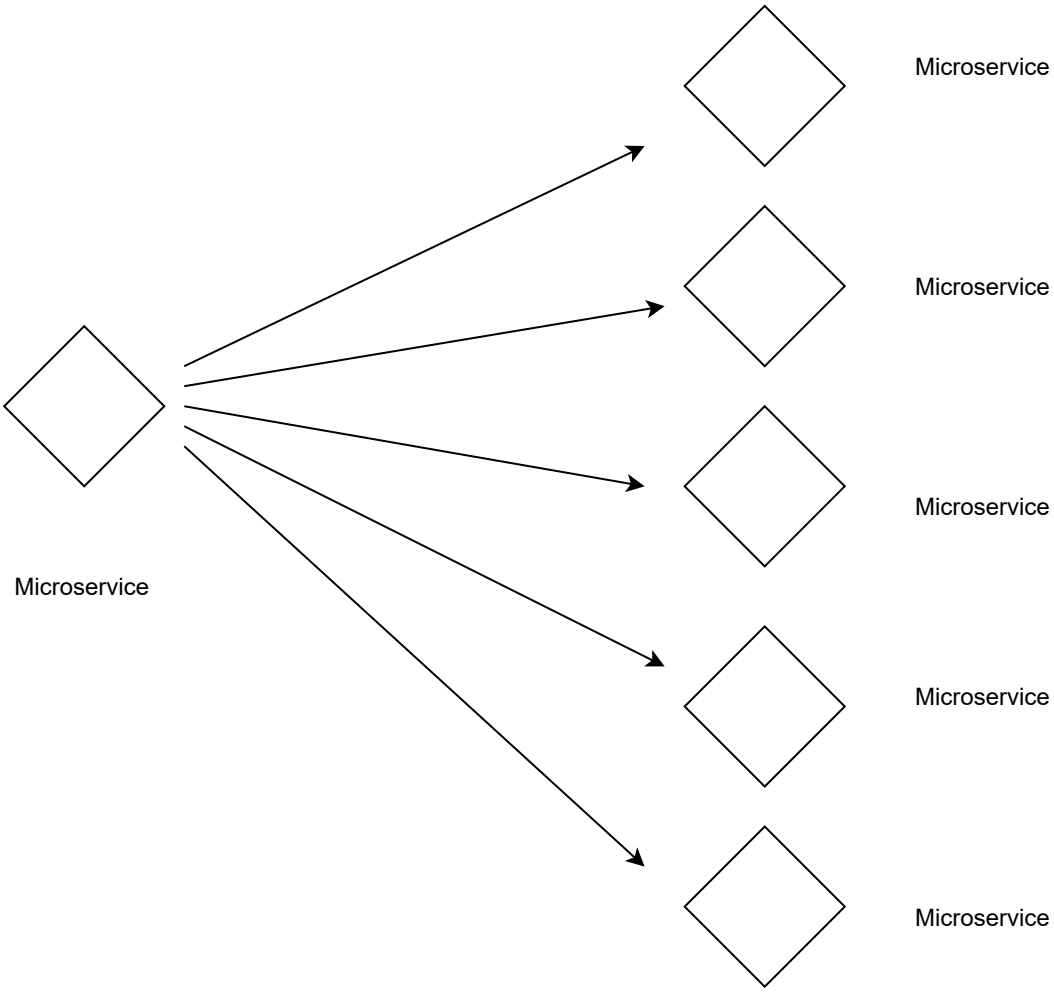
If the microservices increase in size, at some point they will not deserve the name microservice anymore.



Microservices increase in size

Option 2: Constant size microservices increase in number

To avoid the increase in the size of microservices, it is easy to generate new microservices to keep the size of the individual microservices constant over time.



Constant Size Microservices Increase in Number

Resilience

Each microservice has to be able to deal with the failure of other microservices. This has to be ensured when microservices are implemented.

QUIZ

1 What are the factors that influence the technical Micro Architecture?

In the *next lesson*, we'll discuss Reactive Programming.

Reactive Programming

In this lesson, we'll discuss the concept of reactive programming and its relation to microservices.

WE'LL COVER THE FOLLOWING

- Reactive programming
  - Responsive
  - Resilient
  - Elastic
  - Asynchronous communication
- Reactive programming
- Classical server applications
- Reactive server applications
- Reactive programming and the reactive manifesto
- Reactive programming is not necessary for microservices

Reactive programming

One way to implement a microservice is reactive programming. Oftentimes it is stated that microservices must be implemented with reactive technologies.

This section discusses what *reactive* actually is and determines whether reactive technologies are truly needed for microservices.

Similar to microservices, reactive has an ambiguous definition.

The Reactive Manifesto defines the term “reactive” based on the following characteristics:

Responsive

- **Responsive** means that the system **responds as fast as possible**.

Resilient

- Because of **resilience** the system **remains available** even if parts fail.

Elastic

- The system can deal with different levels of load, for instance by using additional resources. After the load peak subsides the resources are freed again.

Asynchronous communication

- The system uses asynchronous communication (message-driven).

These characteristics are useful for microservices. They pretty much correspond to the features discussed in chapter 2 as essential characteristics of microservices.

At first sight, it seems that microservices, in fact, must be written with **reactive technologies.**

REACTIVE PROGRAMMING

Reactive programming

However, reactive programming means something completely different. This programming concept resembles the data flow. When new data comes into the system, it is processed. A spreadsheet is an example. When the user changes a value in a cell, the spreadsheet recalculates all dependent cells.

Classical server applications

A similar approach is possible for server applications. Without reactive programming, a server application typically processes an incoming request in a thread.

If the processing of the request requires a call to a database, the thread blocks until the result of this call arrives.

In this model, a thread has to be provided for each request that is processed in parallel and for each network connection.

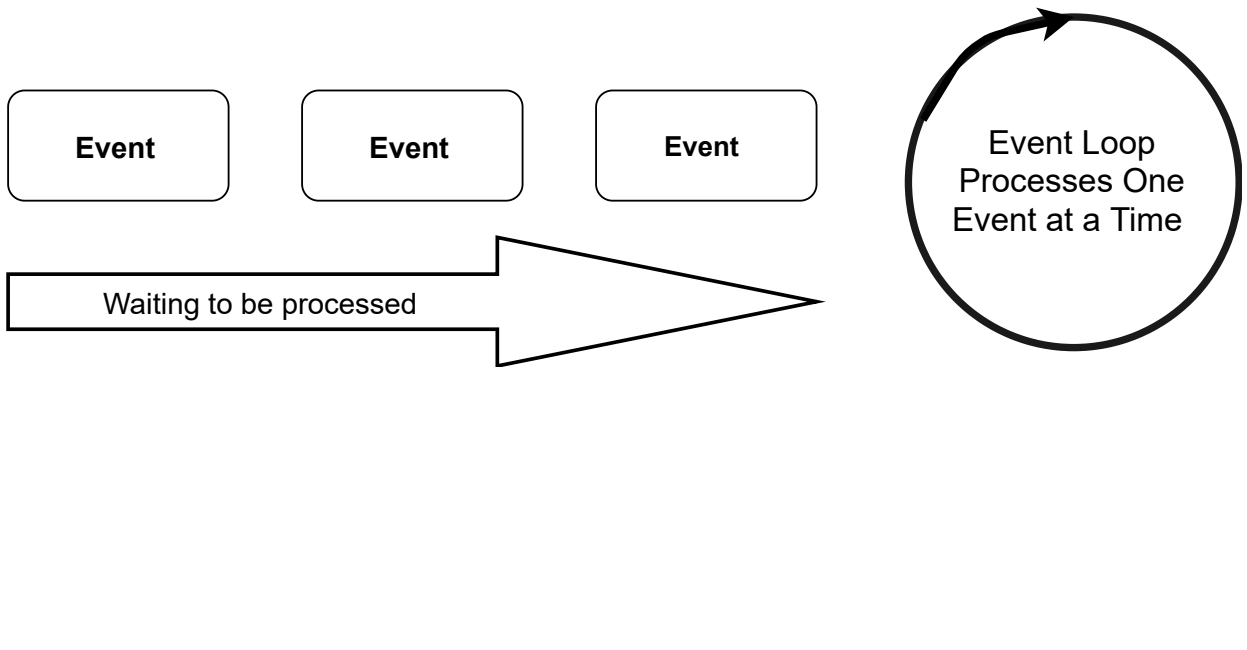
Reactive server applications

Reactive server applications behave very differently. **The application only reacts to events**. It must not block because it is waiting, for instance, for I/O. Thus, an application waits for an event such as an incoming HTTP request.

If a request arrives, the application executes the logic and then sends a call to the database at some point. However, subsequently, the **application does not wait for the result of the call to the database** but suspends processing the HTTP request.

Eventually, the next event arrives, namely the result of the call to the database. The processing of the HTTP request then resumes. In this model, only one thread is needed. It processes the respective current event.

The figure below shows an overview of this approach.



Event Loop

The event loop is a thread and processes one event at a time. Instead of waiting for I/O, the processing of the event is suspended.

Once the results of the I/O operation are available, they are part of a new event which is processed by the event loop.

In this way, a single event loop can process a plethora of network connections. However, processing of the event must not block the event loop for longer unless it is absolutely necessary. Otherwise processing of all events will be

stopped.

THE REACTIVE MANIFESTO

Reactive programming and the reactive manifesto

Reactive programming can support the goals of the Reactive Manifesto:

- **Responsive**: The model can make the application respond faster because fewer threads are blocked. However, whether this really leads to an advantage over a classical application depends on how efficiently the threads are implemented in the system and how efficiently it handles blocked threads.
- **Resilience**: If a service no longer responds, nothing is blocked in reactive programming. This helps with resilience. However, for example, in a classical application, a timeout can avoid a blockage by aborting the processing of the request.
- **Elastic**: With a higher load, more and more instances can be started. This is also possible with the classical programming model.
- **Message-driven**: Reactive programming does not affect the communication between the services. Therefore, communication can or cannot be message-driven in reactive programming as well as in classical applications.

Reactive programming is not necessary for microservices

The Reactive Manifesto is certainly relevant for microservices. But a microservice does not have to be implemented with reactive programming in order to achieve the goals of the Reactive Manifesto.

Whether or not a microservice is implemented with reactive programming can be different for each microservice.

This can be a micro architecture decision and therefore affects only individual microservices, but not the system as a whole.

It is important to understand the difference, because otherwise the choice of technologies might be limited to reactive programming frameworks even though that is not necessary.

It is perfectly fine to stay with established technologies. In fact, using a technology stack that you are used to might be easier and bring faster results.

At the same time, it is possible to try new technologies like reactive programming in one microservice and then use it in other microservices if it has proven to be useful.

QUIZ

1 **Elastic** means that:

COMPLETED 0%

1 of 4

In the *next lesson*, we'll study Spring Boot!

Spring Boot

In this lesson, we'll be starting the discussion about the Spring Boot framework.

WE'LL COVER THE FOLLOWING

- The Spring framework and the Java community
  - Java code
- Compiling the Spring Boot project

The Spring framework and the Java community

The Spring Framework has long been part of the Java community. It has a broad set of features covering most of the technical requirements of typical Java applications. Spring Boot facilitates the use of Spring.

A minimal Spring Boot application can be found in the directory simplest-spring-boot of the project https://github.com/ewolff/spring-boot-demos.

Java code

The Java code from the project shows how Spring Boot can be used.

JavaCode

```
/*
* The Java code from the project shows how Spring Boot can be used.
*/
```

```
@RestController
@SpringBootApplication
public class ControllerAndMain {

@RequestMapping("/") public String hello() {
return "hello\n";
}

public static void main(String[] args) {
    SpringApplication.run(ControllerAndMain.class, args);
}

}
```

Line 5 and 7:

- The annotation `@RestController` means that the class `ControllerAndMain` should process HTTP requests.

Line 6:

- `@SpringBootApplication` triggers the automatic configuration of the environment.

The application thereby starts an environment with a web server and with the parts of the Spring framework that are fitting for a web application.

Line 9:

- The method `hello()`, is annotated with `@RequestMapping`. Therefore it is called upon an HTTP request to the URL `"/"`. The method's return value is returned in the HTTP response.

Lines 13 and 14:

- Finally, the `main()` method starts the application with the help of the class `SpringApplication`.
- The application can simply be started as a **Java application** *even though* it processes **HTTP requests**.

> **Note**: A **web server** is required for handling HTTP in the Java world. It is included in the application.

Compiling the Spring Boot project

Compiling the Spring Boot project

For compiling the project, Spring Boot supports, among others, Maven. Here is a minimal example of a Maven build configuration file:

pom.xml

```
<project>
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.ewolff</groupId>
        <artifactId>simplest-spring-boot</artifactId>
        <version>0.0.1-SNAPSHOT</version>

        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.1.2.RELEASE</version>
        </parent>

        <properties>
                <java.version>10</java.version>
        </properties>

        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
        </build>

</project>
```

The build configuration *inherits settings* from the parent configuration `spring-boot-starter-parent`.

Maven's parent configuration makes it easy to reuse settings for the build of multiple projects.

> The version of the **Maven parent** determines which version of Spring Boot is used.

> The **Spring Boot version** defines the version of the Spring framework and the versions of all other libraries.

Thus, the developer does not have to define a stack with compatible versions of all frameworks, which is otherwise, often a challenge.

QUIZ

1 The version of the Maven parent indirectly determines which version of Spring Boot is used.

COMPLETED 0%

1 of 2

In the *next lesson*, we'll look at how the Spring Boot starter web can serve as a single web dependency.

Spring Boot Starter Web as Single Dependency

In this lesson, we'll look at how the Spring Boot starter web can serve as a single web dependency.

WE'LL COVER THE FOLLOWING

- Spring cloud
- The Maven plugin

The application has a dependency on the library `spring-boot-starter-web`. This dependency integrates the Spring framework, the Spring web framework, and an environment for the processing of HTTP requests.

The **default** for the processing of the HTTP requests is a **Tomcat server** which runs *embedded as part of the application*.

> Thus, the dependency on `spring-boot-starter-web` would be enough as a **sole dependency** for the application!

The dependency on `spring-boot-starter-test` is necessary for **tests**.

> **Note:** The code for the test is not part of this course.

Spring cloud

Spring Cloud is a collection of extensions for Spring Boot which are useful for **cloud applications** and for **microservices**.

Spring Cloud contains **additional starters**. To be able to use the Spring Cloud starters, an entry has to be inserted into the dependency-management section in the `pom.xml` for importing the information about the Spring Cloud starter.

The `pom.xml` files in the examples already contain the required import for this.

The Maven plugin

The Maven plugin `spring-boot-maven-plugin` is necessary to build a **Java JAR** that starts an environment with the Tomcat server and the application.

```
mvn clean package
```

The above command deletes the old build results and builds a new JAR.

> **JAR** is a Java file format which contains **all the code** for an application.

Maven gives this JAR file a name that is derived from the project name. It can be started with:

```
java -jar simplest-spring-boot-0.0.1-SNAPSHOT.jar
```

Spring Boot can also generate **WARs** (web archives) which can be deployed on a Java web server like Tomcat or a Java application server.

QUIZ

1 Why is **Tomcat server** NOT considered to be a dependency of our application?

COMPLETED 0%

1 of 3

In the *next lesson*, we'll look at how Spring Boot fulfills the communication requirement.

Spring Boot for Microservices: Communication

In this lesson, we'll look at how Spring Boot fulfills the communication requirement.

WE'LL COVER THE FOLLOWING

- Communication
  - The importance of SpringMVC in RESTful web services

The suitability of Spring Boot for the implementation of microservices can be decided according to the criteria of this lesson of this chapter.

Communication

For communication, Spring Boot supports **REST**, the previous listing shows. The listing uses the **Spring MVC API**.

THE SPRING MVC API

The Spring MVC framework resides pretty well with REST and provides the necessary API support to implement it seamlessly, with little effort.

The importance of SpringMVC in RESTful web services

I. In Spring MVC, a controller handles requests for all the HTTP methods. This serves as a backbone for RESTful web services.

Example:

- `GET` methods can be used to **handle read operations**
- `POST` methods can be used to **create new** resources
- `PUT` methods can be used to **update resources**
- `DELETE` methods can be used to **remove resources** from the server

II. The representation of data is crucial in REST. This is why Spring MVC allows us to evade `View-based rendering` completely by the use of `@ResponseBody` annotation and many `HttpMessageConverter` implementations. By this, a response can be sent directly to a client.

III. Spring version 4.0's `@RestController` added in the controller class applies message conversations to all handler methods in the controller, preventing the need to annotate each method with the `@ResponseBody` annotation. This also makes our code much cleaner.

IV. Spring MVC also provides `@RequestBody` annotation, which uses `HttpMethodConverter` implementations to convert inbound HTTP data into Java objects passed into a controller's handler method.

V. The Spring framework also provides a template class, the `RestTemplate`, which can consume REST resources. You can use this class to test your RESTful web service or develop REST clients.

These were some of the important features of the Spring MVC framework which assist in developing RESTful web services.

QUIZ

1 What does MVC in the context of Spring Boot stand for?

COMPLETED 0%

1 of 2

In the *next lesson*, we'll be looking at other communication APIs that are supported by Spring.

Other Communication APIs Supported by Spring

In this lesson, we'll look at more communication APIs that the Spring framework can support.

Spring Boot also supports the **JAX RS API**. For **JAX RS**, Spring Boot uses the library **Jersey**. JAX RS is standardized as part of the **Java Community Process (JCP)**.

For messaging, Spring Boot supports the **Java Messaging Service (JMS)**. This is a standardized API that can be used to address different messaging solutions from Java.

Spring Boot has starters for the JMS implementations HornetQ, ActiveMQ and ActiveMQ Artemis. In addition, there is a Spring Boot starter for AMQP. This

protocol is also a standard, but on the network protocol level.

The AMQP starter uses RabbitMQ as an implementation of the protocol. For AMQP as well as for JMS, Spring offers an API that makes it easier to send messages. In addition, simple Java objects (Plain Old Java Objects, POJOs) with no dependencies on any of the APIs can process AMQP and JMS messages with Spring and also return responses to messages.

Spring Cloud offers Spring Cloud Streams for implementing applications for the processing of data streams. This library supports messaging systems such as Kafka, RabbitMQ (see above) and Redis.

Spring Cloud Stream builds on these technologies and extends them with concepts such as streams and therefore goes beyond just simplifying the use of the technology's APIs.

> The integration of technologies in Spring Boot with Spring Boot starters has the advantage that Spring Boot provides the configuration of the environment.

The example Spring Boot application in this chapter uses an infrastructure such as a Tomcat server to handle HTTP requests. This does not require a separate configuration and no additional dependencies. Spring Boot starters also offer such simplifications for messaging and other REST technologies.

Spring Boot applications can also use technologies without a Spring Boot starter. A Spring Boot application can use any technology that supports Java.

> In the end, a Spring Boot project is a Java project and can be extended with Java libraries.

However, it is possible that the configuration can be more complex than a Spring Boot starter.

QUIZ

1

What is Kafka?

COMPLETED 0%

1 of 2

In the next lesson, we'll look at how Spring Boot fulfills the operation requirement for a microservice.

Spring Boot for Microservices: Operation

In this lesson, we'll be looking at how the Spring framework handles the operation of microservices.
Let's begin!

WE'LL COVER THE FOLLOWING

- Operation
  - Deployment in Spring
  - Configuration in Spring
  - Logs in Spring
  - Metrics in Spring

Operation

Spring Boot also has some interesting approaches for operation.

Deployment in Spring

- To deploy a Spring Boot application, it is enough to just copy the **JAR file** to the server and start it. Deploying a Java application can't be further simplified.

Configuration in Spring

- Spring Boot offers numerous options for the configuration. For example, a **Spring Boot** application can read the configuration from a configuration file or from an environment variable. **Spring Cloud** offers support for **Consul** as a server for configurations. The examples in this course use `application.properties` files for configuration because they are relatively easy to handle.

Logs in Spring

- Spring Boot applications can generate logs in many different ways. Usually, a Spring Boot application displays the logs in the console. Output

to a file is also possible. A Spring Boot application can also send the logs as **JSON** data to a central server instead of using a simple human-readable text format. JSON facilitates the processing of log data on this server.

Metrics in Spring

- For metrics, Spring Boot offers a **special starter**, namely the Actuator. After adding a dependency to spring-boot-starter-actuator, the application collects metrics, for example about the HTTP requests. In addition, **Spring Boot Actuator** provides **REST endpoints** under which the metrics are available as JSON documents.

QUIZ

1 A spring boot application can NOT read configuration from _____.

COMPLETED 0%

1 of 5

In the *next lesson*, we'll discuss Spring Boot with regards to resilience and the creation of new microservices.

Stay tuned!

Spring Boot for Microservices: New Microservices & Resilience

In this lesson, we'll be talking about microservice resilience and the creation of new microservices with Spring.

WE'LL COVER THE FOLLOWING

- New microservices
- Resilience

New microservices

Creating a new microservice is very easy with Spring Boot. A **build script** and a **main class** are enough, as shown in the example simplest-spring-boot.

To further simplify the creation of a new microservice, a **template** can be created. The template only needs to be adapted for a new microservice.

> **Settings** for the configuration of the microservices or for logging can be defined in the template.

Thus, a template simplifies the creation of new microservices and facilitates compliance with macro architecture rules.

A particularly easy way to create a new Spring Boot project is to use http://start.spring.io/.

The developer must select the build tool, the programming language, and a Spring Boot version.

In addition, they can select different starters. Based on this, the website then creates a project that can be the basis for the implementation of a microservice.

Resilience

For resilience, a library like **Hystrix** can be useful.

Hystrix implements typical **resilience patterns** such as **timeouts in Java**. Spring Cloud offers an integration and further simplification for Hystrix.

QUIZ

Q Name a library that can be used for resilience in Java.

COMPLETED 0%

1 of 1

In the next lesson, we'll begin our discussion on Go.

Go

In this lesson, we'll be starting our discussion about Go and its relation to microservices.
Let's begin!

WE'LL COVER THE FOLLOWING

- Microservices and the increasing popularity for Go
- Go code
- Go build and compilation
- Docker multi-stage builds
  - Stage 0
  - Stage 1
  - Stage 2

Microservices and the increasing popularity for Go

Go is a programming language that is increasingly being used for microservices due to its great speed and support for concurrency. Concurrency enhances the efficiency of using multiple machines and cores.

Go also provides a powerful standard library for the creation of web services.

For further details on how Go compares with the other 4 languages commonly used for implementing Microservices, visit this site.

Similar to Java, Go is based on the programming language C. However, in many areas Go is fundamentally different from C.

Go code

The Go program below responds to HTTP requests with HTML code.

main.go

```
package main

import (
  "fmt"
  "log"
  //"time"
  "net/http"
)

func main() { http.Handle("/common/css/",
   http.StripPrefix("/common/css/",
   http.FileServer(http.Dir("/css"))))

  http.HandleFunc("/common/header", Header)
  http.HandleFunc("/common/footer", Footer)
  http.HandleFunc("/common/navbar", Navbar)
  fmt.Println("Starting up on 8180")
  log.Fatal(http.ListenAndServe(":8180", nil))
}

// Header and Navbar left out

func Footer(w http.ResponseWriter, req *http.Request) {
  fmt.Fprintln(w,
   `<script src="/common/css/bootstrap-3.3.7-dist/js/bootstrap.min.js" />`)
}
```

Line 3:

The key word `import` imports some libraries, among others for **HTTP**.

Line 10, and 23:

The program's `main` function defines which methods should respond to which **URLs**. For example, the method `Footer` **(line 23)** returns HTML code.

On the other hand, for the URL `/common/css` **(line 10)** the application delivers content from files.

> It is also very easy to implement a **REST** service with **Go**.

In addition, libraries like Go kit offer many more functionalities to implement microservices.

Go build and compilation

Go compilers are particularly well suited for Docker environments because they can create **static binaries**.

> **Static binaries** do not require any further dependencies or a specific Linux distribution.

However, the applications must be compiled to **Linux binaries**. This requires a Go environment that can create Linux binaries.

Docker multi-stage builds

The example uses Docker multi stage builds. Such a build divides the build process of the Docker image into several stages.

First Stage

The **first stage** can compile the program in a Docker container with a Go build environment.

Second Stage

The **second stage** can execute the Go program in a Docker container as a runtime environment that contains only the compiled program.

Consequently, the runtime environment has no build tools and is therefore much smaller.

Docker multi stage builds are not very complicated, as a look at the *Dockerfile* shows:

Dockerfile

```
FROM golang:1.8.3-jessie

COPY /src/github.com/ewolff/common /go/src/github.com/ewolff/common
WORKDIR /go/src/github.com/ewolff/common
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o common .

FROM scratch
COPY bootstrap-3.3.7-dist /css/bootstrap-3.3.7-dist
COPY --from=0 /go/src/github.com/ewolff/common/common /
```

```
ENTRYPOINT ["/common"]
CMD []
EXPOSE 8180
```

Stage 0

Line 1

The **base image** `golang` contains the Go installation.

Line 3

The Go source code is `copied` and compiled into this image **(line 4/5)**. With that, stage 0 of the build is finished.

Stage 1

Stage 1 creates a new Docker image.

Line 7

The image, `scratch`, is an empty Docker image.

Line 8 and 9

The Dockerfile copies the `bootstrap` library (line 8) and the compiled Go binary from stage 0 (line 9) into this image.

The option `--from=0` indicates that the file common originates from **stage 0** of the Docker build.

Stage 2

Line 10

Finally, `ENTRYPOINT` defines the binary that is supposed to be started.

Line 11

`CMD` indicates that no options are to be passed to the binary at the start.

Normally, `ENTRYPOINT` would be a shell that starts the process that is configured with `CMD`. However, in the scratch image, there is no shell.

Line 12

According to Docker Documentation, “The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP. The default is TCP if the protocol is not specified.” So, port 8180 is specified here.

QUIZ

1 What are the advantages of using multistage builds to compile Go code?

COMPLETED 0%

1 of 3

In the *next lesson*, we’ll discuss the potential of Go Lang in the implementation of microservices.

Go for Microservices?

In this lesson, we'll see how Go fits for usage in the implementation of Microservices according to the criteria specified in the Requirements lesson.
Let's Begin!

WE'LL COVER THE FOLLOWING

- Communication
- Operation
  - Deployment
  - Configuration
  - Logs
  - Metrics
- New microservices
- Resilience

The criteria from the second lesson of this chapter for the implementation of microservices can serve as a basis to assess Go's suitability as a microservices programming language.

Communication

Go supports **REST** in the standard libraries. Libraries are also available for messaging systems such as **AMQP**, for example https://github.com/streadway/amqp.

There is also a library for messaging with Redis.

Due to the widespread use of Go, there is hardly any communication infrastructure that does not support Go.

Operation

Go also offers many options for operation.

Deployment

- The **deployment** in a Docker container is very easy with Docker multi stage builds, as already illustrated.

Configuration

- Libraries like Viper support the **configuration** of Go applications. This library supports formats such as **YAML** or **JSON**.

Logs

- Go itself already offers support for **logs**. The Go microservices framework Go Kit contains additional features for logs in more complex scenarios.

Metrics

- For **metrics**, Go Kit supports a plethora of tools such as Prometheus, but also **Graphite** or **InfluxDB**.

New microservices

For a new microservice, it is enough to create the Docker build and then write the source code.

Resilience

Go Kit contains an implementation of resilience patterns such as Circuit Breaker. In addition, there is a port of the Hystrix library for Go.

Microservices have to **communicate** with *other microservices*. This requires a **UI integration** in the **web UI** or **protocols** such as **REST** or **messaging**.

> It is a *macro architecture decision* which communication protocol is used (see chapter 2).

QUIZ

1 What is `viper`?

COMPLETED 0%

1 of 3

In the *next lesson*, we'll discuss variations in the implementation of Microservices.

Stay tuned!

Variations

In this lesson, we'll see how implementation of Microservices can vary.

WE'LL COVER THE FOLLOWING

- Alternatives to Spring Boot

The technical micro architecture decisions can be made differently for each microservice. But there is a connection with the macro architecture.

> The **uniformity** of the operational aspects can be enforced by the **macro architecture**.

If you want to implement a microservice with other technologies in a **Spring Boot** microservices architecture, this can lead to a lot of effort.

A macro architecture decision could be to read out configurations from an `application.properties` file.

This decision does not restrict the choice of implementation technologies. But for a Spring Boot application, the implementation is very simple because this mechanism is built into Spring Boot and the default for Spring Boot applications.

A **Go** application, on the other hand, would have to be adapted to this requirement.

This effect supports a uniform choice of technology for the microservices because implementing a microservice with Spring Boot is easier, therefore, developers would prefer Spring Boot.

A uniform choice of technology has further advantages. For example, developers are more likely to find their way around in other microservices,

and developers of different microservices can help each other out with technology issues.

In order to really treat other technologies as equal, a different macro architecture decision should be made.

Spring Boot offers many more options.

For example, the configuration can be stored in environment variables, transferred via the command line or read from a configuration server.

Alternatives to Spring Boot

In the Java area, there are some alternatives to Spring Boot.

- A classic **Java EE application** with an application server or a web server is also conceivable as an implementation for a microservice. However, in this case, deployment is more complex because the application server has to be installed additionally. Also, application servers and applications must be configured, in some cases even with two different technologies. There are many doubts about the usefulness of application servers.

- thorntail provides a simple JAR deployment. However, instead of Spring APIs it implements the standardized Java EE APIs and supplements them with technologies from the microservices area such as Hystrix.
- Dropwizard has long been offering the possibility of developing Java REST services and deploying them as JARs.

Of course, there are many other possible choices for the programming language apart from **Java** or **Go**.

It is impossible to even list them in this course.

Actually, the point this course makes is that the technologies for the implementation of each microservice are not that important.

It is easily possible to implement each microservice with a different programming language and framework, so the decision can easily be changed.

However, it is much harder to change the technologies for communication, integration, and operations that this course focuses on.

The criteria from lesson 2 of this chapter are a yardstick to check the technologies for their suitability for microservices, as lesson 8 does for Spring Boot and lesson 12 for Go. Such an assessment is recommended for each technology used.

In the *next lesson*, we'll discuss the advantages of microservices that arise from our discussion in the previous lessons, and then formally conclude this course!

Chapter Conclusion

In this lesson, we'll formally conclude this chapter with some notes regarding the advantages of microservices

WE'LL COVER THE FOLLOWING

- Advantages of microservices
- Where to go from here?

Individual microservices can differ greatly in their technical micro architecture. This freedom is a major advantage of microservices architectures.

Advantages of microservices

- The **macro architecture** and the challenges associated with implementing microservices can be used to **derive requirements** for micro architecture and microservices technologies.
- **Reactive programming** can be used to implement microservices, but this is not mandatory to meet the requirements.
- **Spring Boot** and Java meet the requirements, just like **Go** does with the appropriate libraries.
- There are also many other **alternative languages** that can be used to implement microservices, e.g. Python.

Since each microservice can use a different microarchitecture and other technologies, the technical decisions at this level are not so important. They can be revised in any microservice.

Where to go from here?

Thank you for taking this course! We hope you learned the basics of

microservice architecture and the many advantages it has. To continue down this path, you can take the next course in this series.

The next chapter consists of an appendix of instructions for installing Docker locally!

Docker Installation and Docker Commands

In this lesson, we'll look at a quick Docker installation guide.

WE'LL COVER THE FOLLOWING

- Starting Off
- Overview
- Docker Machine Drivers
- Advantage: Separate Environments and Docker on Servers

Starting Off

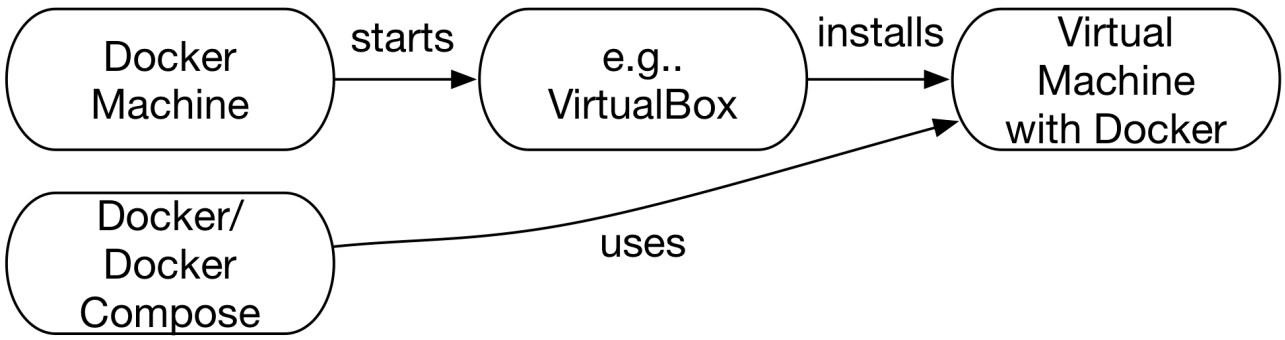
Docker Machine is a tool that can install Docker hosts. From a technical point of view, the installation is quite easy to do. Docker Machine loads an ISO CD image with boot2docker from the Internet.

boot2docker is a Linux distribution and that provides an easy way to run Docker containers. After that, the Docker Machine starts a virtual machine with this boot2docker image.

Particularly convenient with Docker machine is the fact that **using Docker containers on external Docker hosts is just as easy as using local Docker containers**. The Docker command-line tools only need to be configured to use the external Docker host. Afterward, the use of the Docker host is transparent.

Overview

The figure below shows an overview of Docker Machine. Docker Machine installs a virtual machine on which Docker is installed. Docker and other tools, such as Docker Compose then can use this virtual machine as if it were the local computer.



Docker Machine

The command:

```
docker-machine create --driver virtualbox dev
```

creates a Docker host with the name `dev` with the virtualization software Virtualbox. This requires that Virtualbox be installed on the computer.

Afterwards,

```
eval "$(docker-machine env dev)"
```

on **Linux/macOS** configures Docker in such a way that the `docker` command line tools use the Docker host in the virtual Virtualbox machine. If necessary, the shell used must be specified.

```
eval "$(docker-machine env --shell bash dev)"
```

For **Powershell on Windows**, the command is:

```
docker-machine.exe env --shell powershell dev
```

and for **cmd.exe on Windows**, it is:

```
docker-machine.exe env --shell cmd dev
```

`docker-machine rm dev` deletes the Docker host again.

Docker Machine Drivers

Docker Machine Drivers

Virtualbox is only one option. There are many more Docker Machine drivers for cloud providers such as Amazon Web Services (AWS), Microsoft Azure, or Digital Ocean.

In addition, there are drivers for virtualization technologies such as VMware vSphere or Microsoft Hyper-V. Using any of these, Docker Machine can easily install Docker hosts on many different environments.

Advantage: Separate Environments and Docker on Servers

Docker Machine allows one to completely separate Docker systems from each other so that, for example, after a test, nothing remains on the system and all resources are indeed released again. In addition, Docker containers can thus be started very easily on a cloud or virtual infrastructure.

Running the examples in this course directly with Docker is the easiest option and therefore recommended. Docker Machine should be used for the examples only if they are to run on a server or can be completely separated from other Docker installations.

Docker and Docker Compose Commands

In this lesson, we'll study a few Docker and Docker Compose commands.

WE'LL COVER THE FOLLOWING

- Docker Compose
- Docker
- State of a container
- Lifecycle of a container
- Docker images
- Cleaning up
- Troubleshooting

Docker Compose is used for the coordination of multiple Docker containers. Microservices systems usually consist of many Docker containers. Therefore, it makes sense to start and stop the containers with Docker Compose.

Docker Compose

Docker Compose uses the file `docker-compose.yml` to store information about the containers. The Docker documentation explains the structure of this file. The Docker Compose lesson contains an example of a Docker Compose file.

Upon starting, `docker-compose` outputs all possible commands. The most important commands for Docker Compose are:

- `docker-compose build` generates the Docker images for the containers with the help of the `Dockerfiles` referenced in `docker-compose.yml`.
- `docker-compose pull` downloads the Docker images referenced in `docker-compose.yml` from Docker hub.
- `docker-compose up -d` starts the Docker containers in the background.

Without `-d` the containers will start in the foreground so that the output

of all Docker containers happens on the console. It is not particularly clear which output originates from which Docker container. The option `--scale` can start multiple instances of a service, e.g. `docker-compose up -d --scale order=2` starts two instances of the order service. The default value is one instance.

- `docker-compose down` stops and deletes the containers. In addition, the network and the Docker file systems are deleted.
- `docker-compose stop` stops the containers. Network, file systems and containers are not deleted.

Docker

At startup, `docker` outputs all valid commands without parameters.

Tip: Tab-pressing completes names and IDs of containers and images.

Here is an overview of the most important commands. The container `ms_catalog_1` is used as an example.

State of a container

- `docker ps` displays all running Docker containers. `docker ps -a` also shows stopped Docker containers. The containers like the images have a hexadecimal ID and a name. `docker ps` outputs all this information. For other commands, containers can be identified by name or hexadecimal ID. For the example in this course, the containers have names like `ms_catalog_1`. This name consists of a prefix `ms` for the project, the name of the service `catalog` and the sequence number `1`. The name of the container is often confused with the name of the image (e. g. `ms_catalog`).
- `docker logs ms_catalog_1` shows the previous output of the container `ms_catalog_1`. `docker logs -f ms_catalog_1` also displays all other outputs that the container still outputs.

Lifecycle of a container

- `docker run ms_catalog --name="container_name"` starts a new container

with the image `ms_catalog`, which gets the name `container_name`. The parameter `--name` is optional. The container then executes the command that is stored in the `CMD` entry of the `Dockerfile`. But you can execute a command in a container with `docker run <image> <command>`. `docker run ewolff/docker-java /bin/ls` executes the command `/bin/ls` in a container with the Docker image `ewolff/docker-java`. So the command displays the files in the root directory of the container. If the image does not yet exist locally, it is automatically downloaded from the Docker hub on the Internet. When the command has been executed, the container shuts itself down.

- `docker exec ms_catalog_1 /bin/ls` executes `/bin/ls` in the running container `ms_catalog_1`. Thus, with these commands you can start tools in an already running container. `docker exec -it ms_catalog_1 /bin/sh` starts a shell and redirects input and output to the current terminal. This way, you have a shell in the Docker container and can interactively work with the container.

- `docker stop ms_catalog_1` stops the container. It first sends a SIGTERM so that the container can shut down cleanly, and then a SIGKILL.

- `docker kill ms_catalog_1` terminates execution of the container with a SIGKILL, but the container is still there.

- `docker rm ms_catalog_1` permanently deletes the container.

- `docker start ms_catalog_1` starts the container again that was stopped before. As the data is not deleted when the container was stopped, all data is still available.

- `docker restart ms_catalog_1` restarts the container.

Docker images

- `docker images` displays all Docker images. The images have a hexadecimal ID and a name. For other commands, images can be identified by both mechanisms.

- `docker build -t=<name> build <path>` creates an image with the name, `name`. The `Dockerfile` has to be stored in the directory `path`. When no version is indicated, the image gets the version `latest`. As an alternative,

the version can also be indicated in the format `-t=<name:version>`. The Dockerfiles lesson describes the format of the `Dockerfiles`.

- `docker history <image>` shows the layers of an image. For each layer, the ID, the executed command and the size of the layer are displayed. The image to be displayed can be identified by its name if there is only one version of the image with that name. Otherwise, the name and version must be specified via `name:version`. Of course, you can also use the hexadecimal ID of the image.

- `docker rmi <image>` deletes an image. As long as a container is still using the image, it cannot be deleted.

- `docker push` and `docker pull` store Docker images in a registry or load them from a registry. If no other registry is configured, the public Docker hub is used.

Cleaning up

There are several commands to clean up the Docker environment.

- `docker container prune` deletes all stopped containers.

- `docker image prune` deletes all images that do not have a name.

- `docker network prune` deletes all unused Docker networks.

- `docker volume prune` deletes all Docker volumes which are not used by a Docker container.

- `docker system prune -a` deletes all stopped containers, all unused networks, and all images which are not used by at least one container. So, all that remains is what the currently running containers need.

Troubleshooting

If an example does not work:

- Are all containers running? `docker ps` displays the running containers, `docker ps -a` also shows the terminated ones.

- Logs can be displayed with `docker logs`. This also works for terminated containers. The term `Killed` in the logs denotes that too little memory is available. Under Windows and macOS you can find the settings for this in the Docker application under Preferences/ Advanced. Docker should have about 4 GB assigned.
- In case of more complex problems, you can start a shell in the container with `docker exec -it ms_catalog_1 /bin/sh` and examine the container more closely.

Acknowledgements

I would like to thank everybody who discussed microservices with me, who inquired about them, or worked with me on this course. Unfortunately, these folks are far too numerous to name individually. The exchange of ideas is enormously helpful and also fun!

Many of the ideas and their implementation would not have been possible without my colleagues at INNOQ. I would especially like to thank Alexander Heusingfeld, Christian Stettler, Christine Koppelt, Daniel Westheide, Gerald Preissler, Hanna Prinz, Jörg Müller, Lucas Dohmen, Marc Giersch, Michael Simons, Michael Vitz, Philipp Neugebauer, Simon Kölsch, Sophie Kuna, Stefan Lauer, and Tammo van Lessen.

Also, Merten Driemeyer and Olcay Tümce provided important feedback.

Finally, I would like to thank my friends and family, whom I may have neglected while writing this course – especially my wife. She also did the translation into English.

Of course, my thanks goes out to the people who developed the technologies which I introduce in this course and thereby created the foundation for microservices.

I would also like to thank the developers of the tools of https://www.softcover.io/ and Leanpub.