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

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.

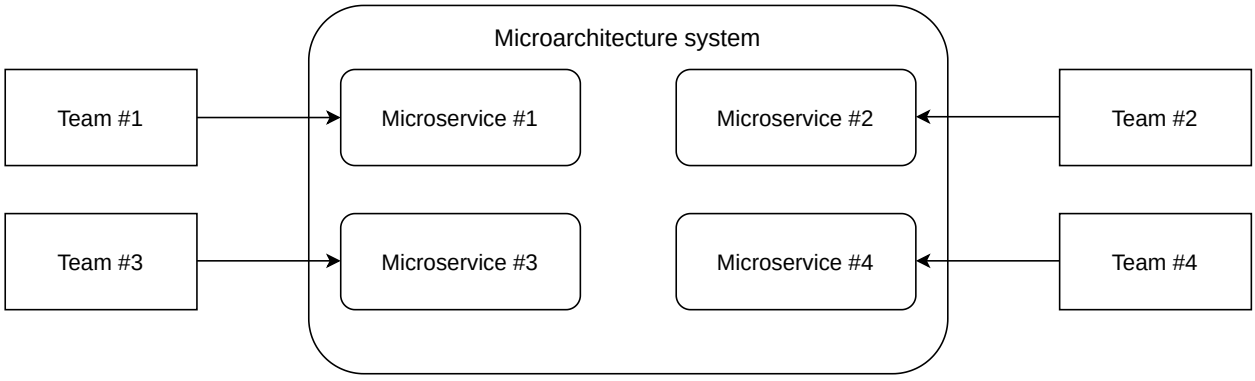
- For example, the **teams** responsible for an individual microservice **can make most technology decisions on their own**.
- When the microservices are delivered as Docker containers, **each Docker container only has to offer an interface** for the other

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 (https://www.educative.io/collection/page/10370001/6518081205567488/6272204058656768), 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

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 (https://en.wikipedia.org/wiki/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,

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 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.

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 (https://www.hello2morrow.com/products/sonargraph), Structure101 (http://structure101.com/), or jQAssistant (https://jqassistant.org/). Advanced module concepts can also, generate such a firewall. In the Java world, OSGi (https://www.osgi.org/) 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.

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?

A) Because attention is only paid to the code level and not the architecture level

B) Because it is not immediately clear which module a class belongs to

C) Because microservices have clear boundaries due to their interface and to introduce a dependency, they will have to call it

Submit Answer

Reset Quiz

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

← Back

Introduction

Next →

Advantage: Continuous Delivery

Mark as Completed

Report an Issue