**Microservices**

**introduction**

We are living in a digital world where almost everything can be done online. The Internet makes it possible to quickly find information, communicate with people around the world, manage your finances, shop from home, listen to music, watch videos, and much, much more.

[Amazon](https://www.amazon.in/), one of the most popular online stores, uses Microservices in java to deliver products to customers decomposed by use case. Spotify uses decomposition by resources to manage the user’s account. So you might be wondering what decomposition by use case or resource means? Just hold onto these terms for a while and we shall discuss them.

**What are Microservices?**

Microservices are small, autonomous programs that function as data producers and data consumers, particularly between service boundaries within a virtualized cloud environment. 100-200 individual microservices might be used to render a single [Amazon](https://www.codingninjas.com/codestudio/problem-lists/top-amazon-coding-interview-questions) web page. Microservices in [Java](https://www.codingninjas.com/courses/online-java-course) are also used by Netflix, [Uber](https://www.codingninjas.com/codestudio/problem-lists/top-uber-coding-interview-questions) web pages.

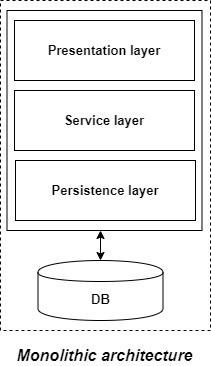
A *microservice* is a small, loosely coupled distributed service. Microservice Architectures evolved as a solution to Monolith architectures’ scalability and innovation challenges (Monolith applications are typically huge – more than 100, 1000 lines of code). It allows you to take a large application and decompose or break it into easily manageable small components with narrowly defined responsibilities.

Microservices in J- also known as the *Microservice architecture* – is an architectural style that structures an application as a collection of the following capabilities:

* Highly maintainable and testable
* Loosely coupled
* Independently deployable
* Organised around the business

In order to understand microservices, we need to understand what are monolithic applications and what led us to move from monolithic applications to microservices in recent times.

**Monolithic applications**   
If all the functionalities of a project exists in a single codebase, then that application is known as monolithic application. We all must have designed a monolithic application in our lives in which we were given a problem statement and were asked to design a system with various functionalities. We design our application in various layers like presentation, service and persistence and then deploy that codebase as single jar/war file. This is nothing but a monolithic application where “mono” represents the single codebase containing all the required functionalities.



But if we were using monolithic applications already then what led us to microservices?

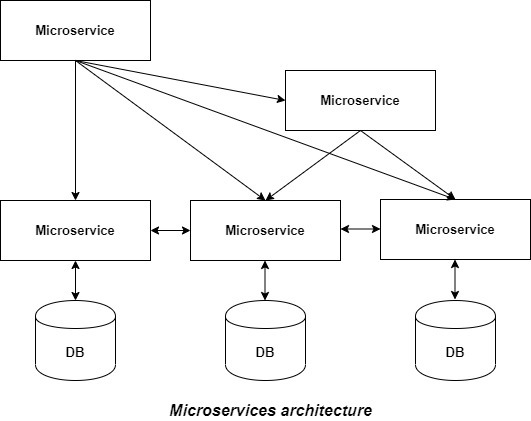
**Disadvantages of Monolithic applications:**

* It becomes too large in size with time and hence, difficult to manage.
* We need to redeploy the whole application even for a small change.
* As the size of the application increases, its start-up and deployment time also increases.
* For any new developer joining the project, it is very difficult to understand the logic of large Monolithic application even if his responsibility is related to a single functionality.
* Even if a single part of the application is facing a large load/traffic, we need to deploy the instances of the whole application in multiple servers. It is very inefficient and takes up more resources unnecessarily. Hence, horizontal scaling is not feasible in monolithic applications.
* It is very difficult to adopt any new technology which is well suited for a particular functionality as it affects the whole application, both in terms of time and cost.
* It is not very reliable as a single bug in any module can bring down the whole monolithic application.

**Advantages of monolithic applications:**

* Simple to develop relative to microservices where skilled developers are required in order to identify and develop the services.
* Easier to deploy as only a single jar/war file is deployed.
* Relatively easier and simple to develop in comparison to microservices architecture.
* The problems of network latency and security are relatively less in comparison to microservices architecture.

**Microservices**   
It is an architectural development style in which the application is made up of smaller services communicating with each other directly using lightweight protocols like HTTP. According to Sam Newman, “Microservices are the small services that work together.”



The Microservice architecture has a significant impact on the relationship between the application and the database. Instead of sharing a single database with other microservices, each microservice has its own database. It often results in duplication of some data but having a database per microservice is essential if you want to benefit from this architecture as it ensures loose coupling. Another advantage of having a separate database per microservice is that each microservice can use the type of database best suited for its needs.

**Principles of microservices:**

* **Single responsibility:** It is one of the principles defined as a part of SOLID design pattern. It states that a single unit, either a class, a method, or a microservice should have one and only one responsibility. Each microservice must have a single responsibility and provide a single functionality. You can also say that: the number of microservices you should develop is equal to the number of functionalities you require. The database is also decentralized and generally, each microservice has its own database.
* **Built around business capabilities:** In today’s world, where so many technologies exist, there is always a technology which is best suited for implementing a particular functionality. But in monolithic applications, it was a major drawback, as we can’t use different technology for each functionality and hence, need to compromise in particular areas. A microservice shall never restrict itself from adopting appropriate technology stack or backend database storage which is most suitable for solving the business purpose i.e. each microservice can use different technology based on business requirements.
* **Design for failure:** Microservices must be designed with failure cases in mind. Microservices must exploit the advantage of this architecture and going down of one microservice should not affect the whole system, other functionalities must remain accessible to the user. But this was not the case in the Monolithic applications, where failure of one module leads to downfall of the whole application.

**Service oriented architecture (SOA) vs Microservices architecture:**

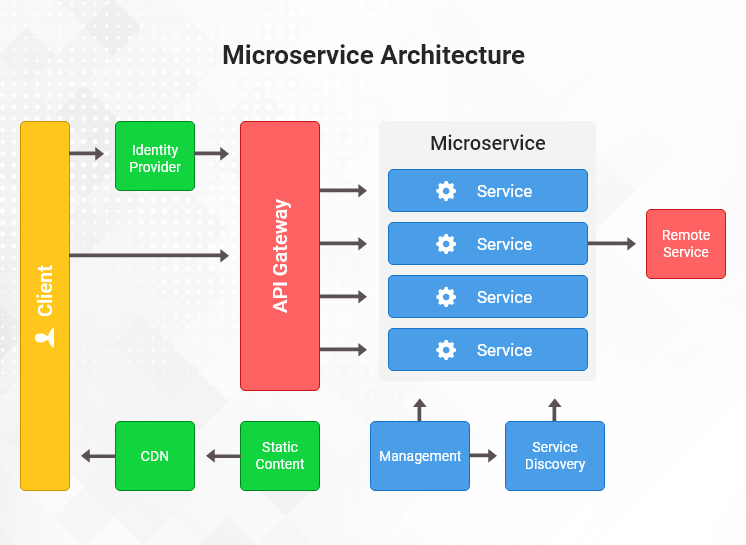
Steve Jones, MDM at Capgemini once said, “Microservices is SOA, for those who know what SOA is”. So, those who know about SOA, mostly think that they are same or the difference is not much clear in their mind. We can’t blame them also, if we talk about a cake and a pastry, we will find more similarities than differences. So let’s try to understand the differences between the two.   
SOA evolved in order to deal with the problems in the monolithic architecture and became popular in the early 2000s. In SOA, the large application is split up into multiple smaller services that are deployed independently. These services doesn’t communicate with each other directly. There used to be an Enterprise Service Bus (ESB, a middleware or server with the help of services using different protocols or message standards can communicate with each other easily) where these services expose themselves and communicate with each other through it. Also, there was no guideline to have independent database for each service.   
Microservices architecture is an evolution of SOA. People also consider SOA as a superset of microservices. In simple terms, microservices is fine-grained SOA. Here, the microservices communicate with each other directly and there is no central dependency for communication such as ESB in SOA. Also, there is a guideline to have a separate database for each microservice. The fundamental idea of evolution of microservices from SOA is to reduce the dependency between the services and make them loosely coupled by above mentioned guidelines.

**Advantages of microservices:**

* It is easy to manage as it is relatively smaller in size.
* If there’s any update in one of the microservices, then we need to redeploy only that microservice.
* Microservices are self-contained and hence, deployed independently. Their start-up and deployment time are relatively less.
* It is very easy for a new developer to on-board the project as he needs to understand only a particular microservice providing the functionality he will be working on and not the whole system.
* If a particular microservice is facing a large load because of the users using that functionality in excess then we need to scale out that microservice only. Hence, microservices architecture supports horizontal scaling.
* Each microservice can use different technology based on the business requirements.
* If a particular microservice goes down due to some bug, then it doesn’t affect other microservices and the whole system remains intact, continues providing other functionalities to the users.

**Disadvantages of microservices:**

* Being a distributed system, it is much more complex than monolithic applications. Its complexity increases with the increase in number of microservices.
* Skilled developers are required to work with microservices architecture which can identify the microservices and manage their inter-communications.
* Independent deployment of microservices is complicated.
* Microservices are costly in terms of network usage as they need to interact with each other and all these remote calls results into network latency.
* Microservices are less secure relative to monolithic applications due to the inter-services communication over the network.
* Debugging is difficult as the control flows over many microservices and to point out why and where exactly the error occurred is a difficult task.



# **How to design microservices architecture?**

Microservices are a holy grail for some and a pitfall for others. A lot of companies are leveraging them as core of their systems, some try to migrate to, some try to migrate away. When you ask someone about their attitude toward this architectural style you'll probably end up with polarised opinions, depending on who you ask.

I'm a Software Architect at SoftwareMill and I've been designing and [consulting](https://softwaremill.com/services/)microservices implementations for the past few years. They are not a perfect solution to every problem, sometimes a well structured monolith is better. How to investigate if your system would benefit from microservices without jumping right away into a binary approach? Here are my lessons learned and thoughts on microservices.

**Containers**

Let's say that you have 50 microservices. Can you imagine a situation in which you manually specify on which machine runs which service? Or one in which you set up different environments versions for different services on production services, e.g. on server A let's install NodeJS 8, on server B NodeJS 12? It wouldn't be pleasant, for sure.

Nowadays an industry standard is to use containers together with a proper orchestration system of automatic deployments and resource management. The most common choice is Kubernetes. It can be self deployed, but not necessarily. Biggest clouds offer hosted k8s environments - [AWS EKS (Elastic Kubernetes Service)](https://aws.amazon.com/eks/), [GCP GKE (Google Kubernetes Engine)](https://cloud.google.com/kubernetes-engine) or [AKS (Azure Kubernetes Service)](https://azure.microsoft.com/en-us/services/kubernetes-service/). This has an additional advantage - if your business decides to move to a different cloud (yes, I have seen such a situation), then the migration will be easier.

Of course the Kubernetes is not the only choice. Some companies are using [DC/OS](https://dcos.io/), [AWS Fargate](https://aws.amazon.com/fargate/) or e.g. [HashiCorp Nomad](https://www.nomadproject.io/).

## CI&CD (Continous Integration & Continous Delivery)

When you have a lot of services, the thing you want to achieve is the lowest ops overhead related to adding a new one. In optimal situations everything should be automated. Continuous integration (CI) which automatically builds code when it lands in the repository. Deploys happening automatically when e.g. commit is marked with a specific tag. For sure no manual jar's copying to production servers!

You need to remember about other topics like log aggregation, monitoring, alerting or tracing. Those are things definitely usable in microservices architecture which will save you when something will go wrong. However, someone needs to maintain them, so it's good to have a DevOps team responsible for that.

## Keep your service stateless

If it is possible, avoid keeping local state with your microservices instances. For sure to avoid down times your goal is to have always more than one instance of the service running. In order to save money, both of them should be active (not active-passive). To do that, they can't just keep anything they want in the memory.

Remember that it does not only affect what you keep in memory, but also how the service processes the data. You may think about situations when there is a single "Business Processor" in the system which operates on incoming data. Try to redesign it from the start, so that it won't become a bottleneck later.

There are some patterns allowing to avoid such situations:

* keep every state in database,
* use distributed caches,
* choose e.g. Akka Cluster with Akka persistence,
* and others.

From the start you have to think about scaling. You won't avoid it, but you can minimize the impact if you design your service while keeping that in mind. There is a big probability that at some point there will be more than 1 instance of the service running.

## Separate Databases

This point is quite controversial. When you have multiple services using a single type of database, let's say Apache Cassandra, it's quite common, to have a single database cluster for the whole system, for all microservices. For sure it allows to lower the operational costs, but also brings drawbacks:

* when one service is using the database more heavily it is often quite difficult to debug which one is doing that. What is more, the traffic it causes can influence the performance of other services.
* when you need to perform some maintenance which will require to shut down the database, then all your microservices are down at the same time.

So what to do? You have to decide by yourself and find the best tradeoff between costs and related possible issues. As a minimum keep tables isolated in separate schemas/keyspaces. This will allow to avoid name conflicts and make migrations to separate clusters in the future easier.

## What are service meshes?

A service mesh is a platform layer on top of the infrastructure layer that enables managed, observable, and secure communication between individual services. This platform layer enables companies or individuals to create robust enterprise applications, made up of many microservices on a chosen infrastructure. Service meshes use consistent tools to factor out all the common concerns of running a service, like monitoring, networking, and security. That means service developers and operators can focus on creating and managing applications for their users instead of worrying about implementing measures to address challenges for every service.

Service meshes are transparent to the application. The service mesh monitors all traffic through a proxy. The proxy is deployed by a [sidecar pattern](https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/#example-1-sidecar-containers) to the microservices. This pattern decouples application or business logic from network functions, and enables developers to focus on the features that the business needs. Service meshes also let operations teams and development teams to decouple their work from one another.

**Logging and Monitoring :**

<https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring>

**When to not Used / Used Microservices**

**<https://containerjournal.com/topics/container-ecosystems/when-to-use-and-not-to-use-microservices/>**