Microservice Architecture

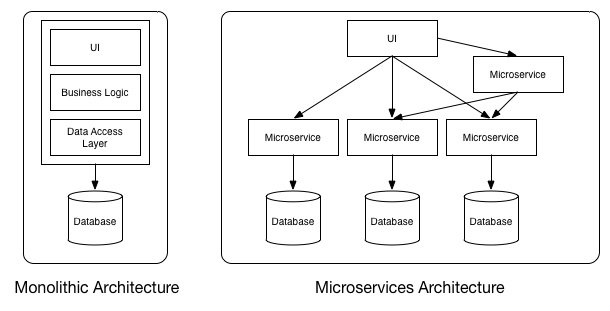
Before going into Microservice architecture, need to know the pain points of Monolithic architecture.

In Monolithic, entire application is developed as a single unit and deployed as single. Any small change in the code may introduce other code which results in application break.

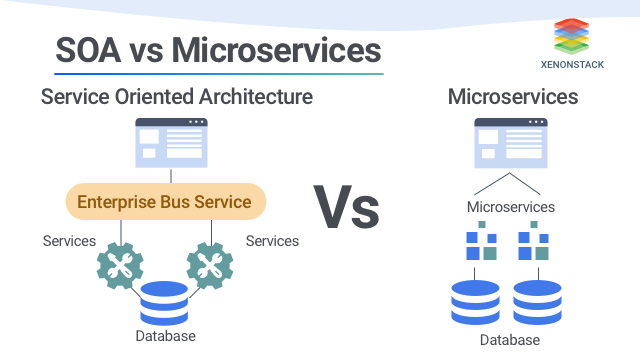
Modules can’t scale independently, will have challenges in continuous deployment.

Even a single module failure brings the application down.

So, to overcome those challenges in monolithic and SOA, microservices comes into picture.



Monolithic vs Microservices Architecture



SOA vs Microservices Architecture

What is Microservice?

It’s an architectural style that develops an application as a set of small services, that are

* Highly maintainable and testable
* Loosely coupled
* Independently deployable
* Organized around business capabilities
* Owned by a small team

Each service runs in its own process. The services communicate with clients, and often each other, using lightweight protocols, often over messaging or HTTP.

What are the problems Microservices trying to solve?

Instead of large teams working on large, monolithic projects, smaller, more agile teams develop the services using the tools and framework they are most comfortable with.

Each of the involved programs is independently versioned, executed and scaled.

These microservices can interact with other microservices and have unique URL’s or names while being always available and consistent even when failures are experienced.

Benefits of Microservices?

* Microservices are small applications that your development teams create independently.
* They can communicate via messaging or http protocols.
* Improved Scalability.
* Fault Isolation.
* Increased Agility.
* Better scope for Technology upgradation.
* Simplified Debugging and Maintenance.

Examples of Microservices Frameworks for Java

Spring Boot, Jersey, Swagger.

Other Microservices examples are, Netflix, Amazon etc.

Characteristics of a Microservice Architecture

1. Componentization via Services:

MSA main intention is to componentize their own software by breaking down into services. Main reason for using services as components is that services are independently deployable.

Componentization, the ability to replace parts of a system, comparing with stereo components where each piece can be replaced independently from the others.

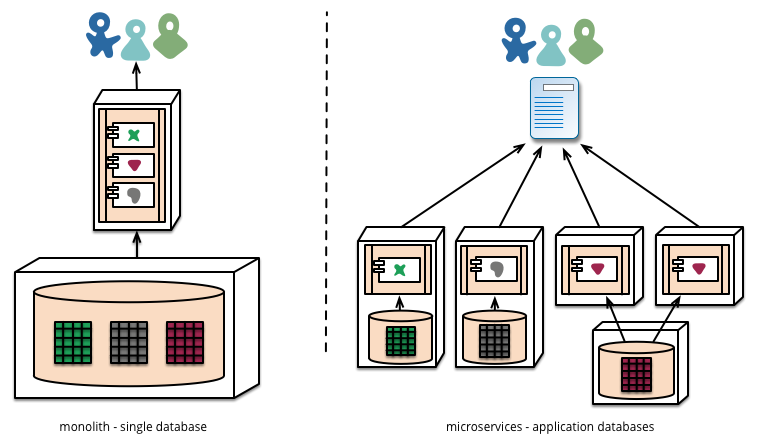
1. Organized around business capabilities instead of around technology.
2. Smart Endpoints and Dump pipes:

Microservices explicitly avoiding the use of Enterprise Service Bus (ESB). Instead, it uses two protocols i.e., HTTP request-response with resource API’s and lightweight messaging.

1. Decentralized Governance.
2. Decentralized Data Management:

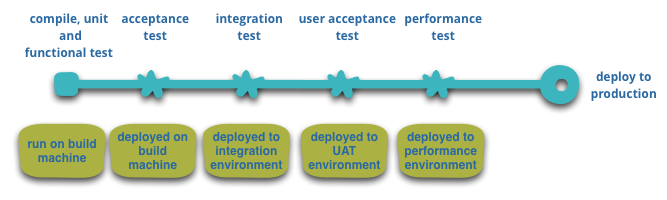
Decentralization of data management represents one database for each service instead of one database for a whole application.

While monolithic applications prefer a single logical database for persistent data. Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems.



1. Infrastructure Automation:

Infrastructure automation with Continuous delivery being mandatory.



Applications developed using Microservice Architecture should achieve CD-CI effectively.

1. Design for Failure:

A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services. Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible. This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it. The consequence is that microservice teams constantly reflect on how service failures affect the user experience.

Microservice teams would expect to see sophisticated monitoring and logging setups for each individual service such as dashboards showing up/down status and a variety of operational and business relevant metrics. Details on circuit breaker status, current throughput and latency are other examples we often encounter in the wild.

1. Evolutionary Design:

Microservice practitioners, usually have come from an evolutionary design background and see service decomposition as a further tool to enable application developers to control changes in their application without slowing down change. Change control doesn't necessarily mean change reduction - with the right attitudes and tools you can make frequent, fast, and well-controlled changes to software.

Putting components into services adds an opportunity for more granular release planning. With a monolith any changes require a full build and deployment of the entire application. With microservices, however, you only need to redeploy the service(s) you modified. This can simplify and speed up the release process.