1. <https://www.scaler.com/topics/software-engineering/microservices-design-pattern/?utm_source=midfunnel&utm_medium=wagroup&utm_campaign=bau-campaign_academy_may-2023-w4_panIndia_all_mc-engagement&utm_content=mc>
2. <https://www.scaler.com/topics/spring-boot/what-is-microservices/?utm_source=midfunnel&utm_medium=wagroup&utm_campaign=bau-campaign_academy_may-2023-w4_panIndia_all_mc-engagement-2&utm_content=mc>

**MICROSERVICES**

* Large applications are divided into small services which are **independent** but are connected with the help of message queue or APIs.
* Each service will serve only **1 business requirement**.
* These services are **designed**, **scalable, deployable independently**.
* It is adopting **service-oriented architecture (SOA)** in which entire program is split into smaller interconnected services.

3 MAJOR GUIDELINES

* **Independent**: Each microservice should be deployable separately.
* **Coupling**: All microservices should be loosely connected such that changes in one do not affect the other.
* **Business Goal:** Each service unit in the application should be the smallest and capable of achieving a single business goal.
* **Own the state –** Each service should have its own independent database. IF other service needs any data they can request the service. It helps in **Polygot persistence** (RDBMS and NoSQL).
* **Share Information using Well-Defined Interfaces like REST MQ etc.**

**MICROSERVICES DESIGN PATTERN**

1. **Decomposition Pattern** – How we will decompose large application into smaller services.

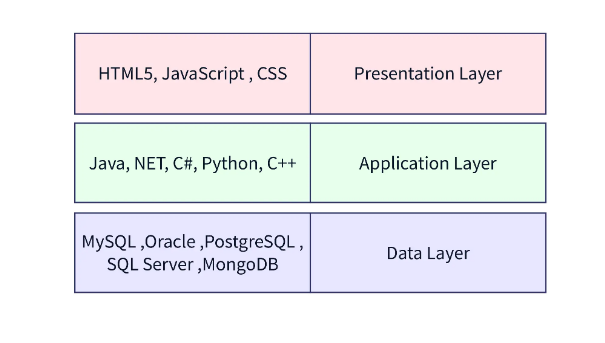
* **Business capability** – Create a service for each business capability.
* **Subdomain** - Decomposing an application using business capabilities is an excellent place to start, but you will run upon so-called "God Classes" that will be difficult to decompose. Several services will share these classes.
* **Strangler** – New services should be create in microservice while older remain in monolith. As development progresses the monolith decreases and micro increase.

1. **Integration Pattern** – How client will access and communicate with each service. They can do so with help of REST or AMQP.

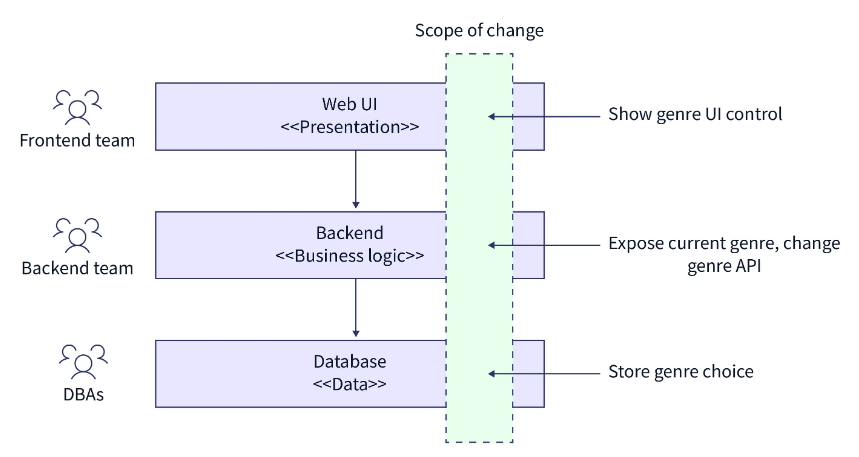
API gateway pattern, Aggregator pattern, Routing pattern etc.

1. **Database Design Pattern** – Whether each service will have its own database or they will use shared databases.
2. **Observability Pattern** – It deals with logging and performance metrics. Each service will produce logs in consistent manner. We use these logs to do troubleshooting and problem investigation.
3. **Cross-cutting concerns –** Service discovery, external config and deployment of these services. These services interact with third-party server (email, message queue, payment etc) and other development servers ( DB servers, service registry).

**3 tier Architecture and Microservices**



* This separation between the three teams explains the reason for having a **three-layer architecture**, where each team is responsible for one layer based on the responsibility.
* The chief benefit of the three-tier architecture is that each tier runs on its infrastructure. It can be developed simultaneously by a separate development team and updated or scaled as needed without impacting the other tiers.
* The problem with 3 tier arch is any new feature addition will require changes in all 3 tier and each team is dependent on another for their development.



Microservices does vertical slicing in 3 tier arch and one team is responsible for developing the feature from UI to backend to data layer.

As the scope of change is spread across all three tiers, this calls for designing the system based on business functionality where each team is end-to-end responsible for delivering specific business functionality

**ADVANTAGES**

* **Scalable** – As services are deployed independently they can be scaled easily.
* **Resiliency** – As fault is contented in one service.
* **Release cycle is smaller**- As services are independent we can develop, demo and release them faster.
* **Less code per service helps in testing(unit testing is easier but integration testing is complex)**.

**DISADVANTAGES**

* **Complexity –** As services are distributed the complexity of distributed system comes. Integration is complex.
* **Monitoring and troubleshooting –** need to develop well logging and monitoring mechanism.
* **Multiple database brings more complexity.**
* **COST**

**Best Practices for Microservices**

* **Use async communication** - To avoid building tightly coupled components, consider using asynchronous communication wherever possible. Asynchronous communication provides loose coupling among services.
* **Maintain backward compatibility** - To deploy changes independently, ensure contract and API changes are backward compatible. When making a breaking change, expose a new version of your endpoint while continuing to support older versions. Consumers can choose to use the new version at their convenience.
* **Version your releases** - Use semantic versioning to convey the type of changes with each release. So that the consumer of your service knows whether a release is major, minor, or patch.
* **Use Circuit breaker to avoid cascading failure** - A circuit breaker is a device whose purpose is to protect electrical equipment from damage in case of a short circuit or overload. The same pattern can be applied to microservices. A software circuit breaker stops sending requests to a service that is not responding or responding promptly so that a faulty service can be isolated, which stops cascading failure to other dependent services.