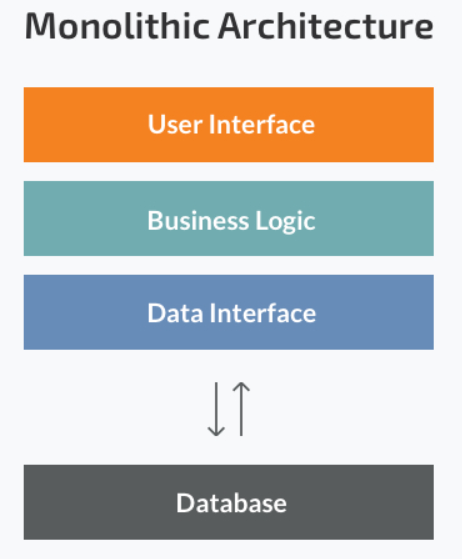
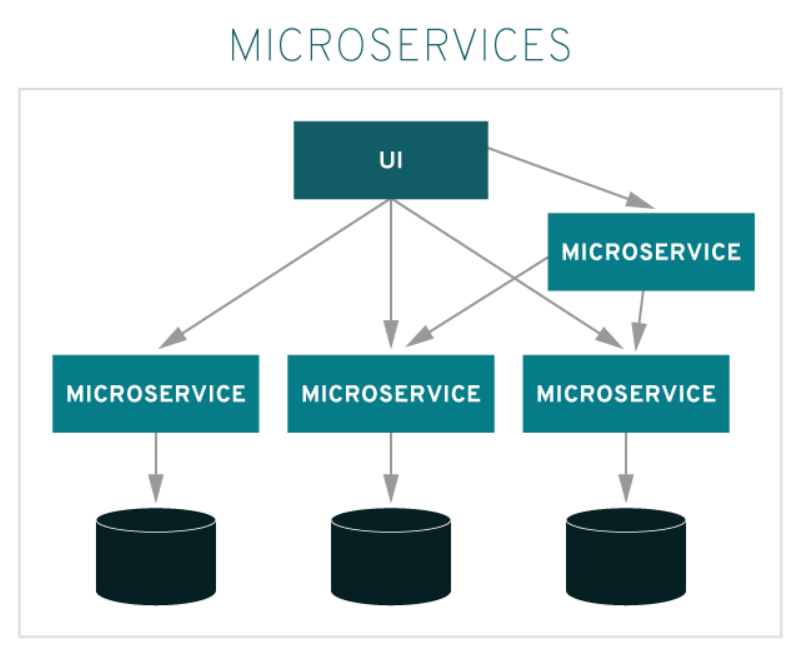
**Monolithic Architecture**

A monolithic architecture is the traditional unified model for the design of a software program. Monolithic, in this context, means composed all in one piece. Monolithic applications are designed to handle multiple related tasks. They’re typically complex applications that encompass several tightly coupled functions.

**Introduction to Microservices**

It is a software architecture pattern. In contrast to the monolithic approach, microservices architecture involves smaller applications deployed independently as loosely coupled services, tied together through application integration. Microservices can be simpler to develop. They’re smaller in scope and therefore smaller in size, which makes it easier for developers to improve them through continuous integration and continuous delivery (CI/CD). They can be written in any programming language. And they can communicate with other microservices through APIs.

1. Microservices is a small unit that has only one responsibility or single logic which solve a specific problem.
2. Microservices are the evolution of service-oriented architecture.
3. Microservices are small and independent services that work together.
4. A style of engineering to build highly automated, independent and evolving software.
5. Each Microservices can be deployed independently.
6. A Microservices itself persist its own data or external state.
7. All services don’t need to share the same technology stack, libraries, or frameworks.

**Microservices Principles**

1. **Modelled around business domain**

Business domain understanding of your project, how many business domains you have in your project.

We will have separate microservices for different domain/objects

For e.g.

- One microservice for Inventory Management

- Second microservice for Order Management

- Third microservice for Account Management

1. **Culture of Automation**

Automation can be achieved with the help of develops (for e.g. Azure, Jenkins)

To Automate the release pipeline.

1. **Hide Implementation Details**

Each microservices can have the different technology stack (for e.g. one microservices is on Java stack other one is one .Net stack etc.)

1. **Decentralize all the things**

Each microservice has its own database, means the tables are centralized in different databases.

1. **Deploy independently**

Each microservice should be deployed independently with the new versioning so it would not affect the existing microservices.

1. **Isolate Failure**

Only the failure microservice will get affected or down it will not make any impact on the other microservices or the application, only the things which are related to the failure microservice will get down for some time.

1. **Highly Observable**

This basically deals with the error logging or tracing to get the error point.

**When to use Microservice Architecture**

* Large application that require a high release velocity.
* Complex application that need to be highly scalable.
* Applications with rich domains or many subdomains.
* An organization that consists of small development teams.

**Advantage of Microservices**

* Language Independent and framework independent.
* Independently develop, deploy, version and scale component services in seconds without compromising the integrity of an application.
* Zero downtime upgrades with devops.
* Support CI/CD
* Easy to integrate with third parties like payment api gateway, sms api gateway etc.
* Small Teams
* Better Fault Isolation keeps other services to work even on failure.
* Scalable

**Disadvantage of Microservices**

* **Complex Networking**
  1. It’s hard to trace where the actual hit will occurs
* **Development and testing:** Building and testing a service that relies on other services need domains understanding and refactoring them can be difficult.
* **Lack of governance:** The decentralized approach to building microservices has advantages, but it also lead to so many problems like maintenance because of many different languages and frameworks.
* **Data Integrity:** Each microservices is responsible for its own data persistence. As a result data consistency can be a challenge.
* **Management:** To be successful with microservices require a mature DevOps culture. Correlated logging across services can be challenging for a single user operation.
* **Versioning:** Be careful while updating a service. It must not break services that depend on it. So without careful design, you might have problems with backward or forward compatibility.
* **Skillset:** Microservices are highly distributed system. So need a skilled and experience team to implement it.
* Overhead of Database and Server
  1. Because each microservice have its own database

**Need of API Gateway**

* Traffic Routing
* Expose unified end point
* API Composition
* Caching
* Logging
* Authentication
* Authorization
* Load balancing
* Service Discovery

**DevOps-** Docker, Git, Jenkins

