# Introduction to Microservice

## Monolithic Architecture

* Both the backend and frontend tightly coupled.
* Easy to deploy.
* Mostly used for small projects and prototypes.
* Incredible hard to scale due to everything relying on things working

## Service Oriented Architecture

* The architecture style of breaking up application into a backend and frontend using services.
* Decoupled the frontend and backend.
* Smoother response time because data is being sent by two different servers.
* Easier to develop into complex projects.

## Microservice Architecture (MSA)

* Implementation of SOA that breaks down services even more (Think of S on SOLID)
* It wants each service to be responsible for one thing only.
* At the end of day, it will make multiple services that must communicate over a network.
* Starting this architecture is complex and making the infrastructure for it but once everything is setup, developing and scaling your project.
* Very popular architecture style for big IT companies.

## MSA Characteristics

## Single Responsibility Principle

* One service must be responsible for one thing.

## Encapsulated

* Each service must be encapsulated the data and behavior into a single unit.
* Each data for each service must be private and can only be seen by its service.
* Therefore, each service will have its own private database that it has access to.

## Independent

* It means that each service can develop on its own without affecting other services.
* Each service can be developed by different type of language.
  + Ex: Python is mostly used for data processing with AI since developing AI is easier

So, a service that uses Python language would make sense to develop.

# Benefits

* Scalability
  + Adding new features to a single service won’t affect your other service.
  + You can have separate teams working on its own service.
* Simplicity of developing new features
  + Once the infrastructure is setup adding new features will be a lot easier and scaling it to have multiple people work on those features is also a lot easier.
* Deployment of individual service
  + Easy to deploy the individual service when scaling out.
* Fault Tolerant
  + One service going down due to some external force or bad coding won’t affect your other service.
  + A nice safety net that one service won’t crash your entire operation for the whole company.
* Language Agnostic
  + You can work with different languages if they all send the data in the same protocol (http)
* Testable
  + Easy to test just your own service instead of worrying about other services.

# Drawbacks

* Deployment of the whole entire environment
  + Deploying everything at the same time (usually at the initial setup) will take awhile and might cause problems.
* Complexity when it comes to communication between services
  + So, with join and aggregation of different data can become very convoluted.
* Monitoring
  + Checking on multiple services to see if there is an issue might take some time.
  + Thankfully there are automated processes that checks our services for us.
* Consistency
  + Data consistency is a bit of a challenge to accomplish
* Communication is key
  + Since there are multiple services and they must have a stable communications or else things fails and data inconsistency might happen.

# Key things to help make MSA work better

## Service Discoverability

* You have service registries that contains information about the services in your MSA ecosystem.
  + Think of a phone book or how DNS works (Domain Name System).
* They will let you know which service is working and which are down.
* (Comes built in K8)

## Gateways

* They are responsible for helping the services communicate with each other.
* It is essentially the hub of communication between services and takes care of the authorization
* So instead of a service talking to each other, a service will just talk to the gateway and the gateway will take care of whatever the service needs and talk to other services to meet that need.

## Load Balancing

* With multiple services running in different containers/servers/nodes/some sort of deployment artifact, you have a load balancer to handle that.
* They will balance the outgoing/internal traffic to the instances of the service.

## Circuit breakers

* If something fails, it’ll make sure that it won’t create a cascading failure of your other services.
* Almost the same as what your electrical circuit breaker does in your home.

## Message Queue

* Another mechanism to help with communication between services.
* It is a form of asynchronous service to service communication
* It stores the messages in a queue until they are processed especially with spiky workload (multiple users suddenly asking for 1000 http request at the same time).
* It is essentially buffer.