# Introduction to Microservices

## Monolithic Architecture

* Both the backend and the frontend are tightly coupled
* Very easy to deploy
* Mostly used for small projects and prototypes
* Very hard to scale due to everything being tightly coupled
  + You can’t evolve frontend separately from the backend and vice versa

## Service Oriented Architecture

* The architecture style of breaking your backend into service that can be used for one or more frontend.
* Decoupled the frontend from the backend.
* Makes it easier to scale your project by having the backend and frontend be developed the same time.
* Easier to develop into more complex projects.

## Microservices Architecture

* Implementation of SOA that breaks down services even more to have their own responsibility and give a team of developers to just develop that one responsibility even further (Think of the S in SOLID).
* Starting this architecture is extremely complex and expensive.
* Very popular architectural style used by big IT companies such as Google, Amazon, Facebook, etc.

# MSA Characteristics

## Single Responsibility Principle

* One service must be responsible for one thing.

## Encapsulated

* Each service must encapsulate the data and behavior as a single unit.
* Each data for each service must be private and can only be seen by its own service.
* Therefore, each service must have each own private database it has accesses to.

## Independent

* Each service must be independent from each other.
* You can develop a service without worrying about breaking the other service that is being developed right now.
* Furthermore, each service can be developed in a completely different language if needed based on the responsibility of that service.

# Benefits

* Scalability
  + You can scale your application depending on how many people you have working at it.
  + It isn’t limited by how you constructed your entire architecture.
* Simplicity of developing new features
  + Once the infrastructure is setup.
  + Adding new features is a lot easier and you don’t have to worry about breaking other features from different services.
* Deployment of individual service
  + Easy to deploy the induvial services when scaling out.
  + Trying to redeploy multiple services at once will take forever to deploy
* Fault Tolerant
  + One service going down due to some external force or bad code won’t affect your other services.
  + A nice safety net that won’t crash or halt your entire operations for your whole company
* Language Agnostic
  + You can work with different languages if they all send the data in the same protocol (HTTP, HTTPS)
  + Enables you to work with the right language for different responsibilities
* Testable
  + Easy to test when each service serves only one purpose

# Drawbacks

* Deployment of a whole entire environment
  + Deploying everything at the same time (usually the initial setup) will take a while and might even cause problems due to deploying a whole ecosystem of services.
* Complexity when it comes to communication between services
  + So, with joins and aggregation of different data can become very convoluted
  + Very simple to develop each service separately but very difficult on making them communicate with each other to do some operation
* Monitoring
  + Checking on multiple services to see if there is an issue might take time (especially when your microservice ecosystem grows)
  + Thankfully there is an automated process that do it 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 communication or else things can fail and data inconsistency might happen
* Overall, it is a steep starting point for using MSA

# Key things to help make MSA work

## Service Discoverability

* You have a service registry that contains information about the services in your MSA ecosystem.
  + Think of a phone book or how DNS (Domain Name System) works
* They are responsible for automating the process of checking/monitory on your services
* They let you know which service is fine and which are down
* (comes built in with K8s)

## Gateway

* They are responsible for helping the services communicate with each other.
* It is essentially the hub of communication between services and user.
* So instead of services talking to each other directly, a service can just talk to a gateway and the gateway will take care of whoever it needs to talk to and accomplish what that service wanted.
* It essentially abstracts the entire MSA ecosystem for end users and make it appear like we are just talking to one service

## Load Balancing

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

## Circuit Breaker

* Almost the same as what your circuit breaker does in your home.
* If something fails, it’ll make sure that it won’t create a cascading failure of your other services.
  + Like your electric circuit breaker that protects your other circuits from damage if another circuit causes a short circuit.

## Message queues

* Another mechanism to help with communications between services.
* It is a form of asynchronous service-to-service communication.
* It stores messages in a queue until they are process individually especially with spike workloads.

# Kubernetes

* Just a technology that helps us create a microservice architecture that has premade tools and features for us to tap on.

## Container Orchestration

* Best way to scale your app in this modern world
* This essentially will automatically deploy more containers depending on the amount of workload that is currently experiencing

## Deployment

* Deploying the containers on a server

## Scaling

* It will have multiple containers running the same image
* Meaning it has the exact copy of whatever published of your image at the time

## Network

* Load balancer and service discoverability is with container orchestration

## Monitoring

* Service health checks
* Checking if the containers are still working
* Can also deploy new containers to replace broken containers

## Summarize

* K8s is a container orchestration tool
* It also gives us many tools to help setup everything and manage them autonomously
* We just must tell it what image(s) to provide, and it will handle the deployment/scale of them

# K8s Architecture

## Containers

* Runnable instance of an image

## Pods

* They are the smallest unit in Kubernetes
* They are made up of one or more containers
* The pod will keep on running and never stopping and if one pod somehow dies a new pod will come up to replace it
* Each pod will have a unique IP address, storage, config information for pod and containers

## Node

* Either a physical or virtual machine that contains necessary services to run multiple pods
* Managed by the control panel that allows container orchestration to take place by managing different pods

## Cluster

* A set of nodes that runs containerize apps (a group of nodes)
  + In a cluster there is a master node and one or more worker nodes

## Master Node

* Controls the state of cluster
* Origin of all task assignments
* What it does:
  + Schedule and scale app
  + Maintain the cluster’s health/state
  + Implements updates

## Kubelet

* Primary node agent
* Registers the node with the master node
* Used by master nodes to monitor other nodes

# Some configuration and cool features in K8s

## State management

* Essentially you have two choices to make of how you want your K8s to operate

### Stateless

* Does not rely on earlier requests information to process other requests
* Server does not hold request information
* Will treat every request like it came from a completely different user

### Stateful

* Request based on the information related with each request
* Information is stored from earlier requests
* Same server must be used to process all the request from the same user

## Secret Management

* K8s object uses secrets to manage secure access
* Like the secrets in Github, this is the more secure way for you to store sensitive information and just have your yaml/config file reference those secrets to get information
  + Ex:

ConnectionStrings

Tokens

Password

Etc.

## Scalability and Failover

* Since K8s has the power and capability to easily scale your application based on demand, it can also deliver that same capability if a pod went down
* It can create a new pod to replace the broken one and load balancer will auto route any traffic going to the old pod to the new pod so the traffic wouldn’t even noticed that something went wrong
* This process is what we call a failover, it is the ability to rely on some backup system automatically and seamlessly if something went wrong

## Ingress

* This is the gateway version of K8s
* It is the middleman between your nodes and your end user and ensures the end user just needs to access one endpoint and will be automatically rerouted to whatever the end user needs to go to
* Unfortunately, it doesn’t come with K8s when you set it up, but you must create an ingress controller yourself
* This is how you will configure your K8s.