Microservices

The parts that make up a whole

# Architectural Styles

## Monolithic Architecture

The architectural style of having everything in one package. UI, BL, DL, all in one deployment. Easy enough to spin up for a prototype, but not ideal for a complex application in production. A little difficult to scale since everything is tightly coupled. When multiple teams/people work on a monolithic application, you can expect a lot of merge conflicts. Having everything in one package does not necessarily mean its simpler, it might make adding new features even more complex. Furthermore, there is the issue of the overhead that comes with tightly coupling the UI with the logic you need to process the data. For scaling, you need to deploy the app (as a whole) to another server even if certain services are not being used as much as the popular ones. In conclusion, this architectural style is perfect for simple applications with a limited number of features. As well as for learning development in a new language.

## Service Oriented Architecture

The architectural style of breaking up your application into services. A service is an independent piece of logic that has a certain responsibility such as: UI, Authorization, BL, etc. The idea of a service in SOA is very broad. You could group together multiple pieces of logic (as long as they follow a theme of functionality). Easy to scale (since you only upscale if you need to). A little more overhead than monolithic apps since the services need to communicate with each other. Overall quicker response time because the data being sent is more lightweight with the decoupling of your logic. Perfect for applications that are complex and used in building for scale.

## Micro Services Architecture

Implementation of SOA that granularizes the idea of a service. In MSA a service embodies the SRP, meaning that each service is responsible for one thing (i.e a resource or some logic) and one thing only. It just makes for a bigger collection of services that must communicate over a network. Of course, there are a couple of development practices involved in successfully executing this architectural style since in its quest for simplicity the execution becomes complex. Very popular architectural style used by Facebook, Amazon, Netflix, and Google.

# MSA Characteristics

## SRP

Single Responsibility Principle. Doing one thing well. Following a certain theme of functionality. For example, a hero service would be responsible for CRUD involving a hero, and a superpower service would be responsible for CRUD involving a superpower involved with a hero.

## Encapsulated

As microservices encapsulate the data (or state) and behavior as a single unit, persistent data for each service must be private. Consumers can only access the data through its published interfaces or APIs.

## Independent

Data of each service is independent from each other. Common practice is to have a separate db for each service. (This leads to certain problems) Furthermore, because each service is independent of each other, you can develop them in different languages. Perfect for tailoring services with languages that provide the best support for certain logic. Like using python for a service that involves data processing with AI.

# MSA benefits

## Scalability

Easier to add new features without affecting existing services. New feature = new service. You can also take down a service without affecting the other services.

## Simplicity

Because services follow SRP, its easy enough to spin up services. They’re usually resource oriented (i.e one resource per service) or logic oriented (a single service follows a functionality theme).

## Deployment

Easy to deploy individual services when scaling out. Because they are travel sized for your convenience.

## Fault Tolerant

When a service goes down, other services are still okay. A service shutting down does not shut down the whole operation. Just like an employee being sick doesn’t shut down the whole company.

## Language Agnostic

Code one service in python, another in java, another in c#, another in JS. They all work together and send messages via http.

## Testable

With SRP, your services are easily testable. You’ve already decoupled the dependencies to another service so the logic should be simple enough to test.

# MSA Drawbacks

## Deployment

Because you have a whole ecosystem of services to deploy, good luck.

## Complexity

Even though the logic in the services themselves are simple, the actual communication between services and aggregation of data becomes convoluted. It can be overwhelming. Complicated routes and calling chains.

## Monitoring

With a collection of services, checking up on them is an issue. Similar to checking up on multiple people constantly, it’s tiring.

## Eventual Consistency

Because its common practice to have separate data storages for each service, you have to communicate changes to the data to the services that are related to that data, eventually. So you’ll eventually get data consistency/integrity.

For example if I decouple my hero service to a hero and superpower service. So the hero service would have some reference to a superpower (that is nullable because the hero exists before the superpower). And the superpower service would have a reference to my hero that owns that superpower. So when you create a hero, you also need to tell the super power service to create a super power, get the reference to that super power, and update the created hero to hold that reference.

## Chattiness

Because services are independent and there’s a lot of them, communication becomes a problem. Too much traffic, too many complicated communication channels.

# MSA Development

## Service Discoverability

You have service registries that contain information about the services in your MSA ecosystem. (Like a phonebook) What these service registries do is automate the monitoring of your services. Check up on each service, make sure they’re still alive, communicate if they’re down.

## Gateways

This helps with the chattiness. A gateway is a service that acts as the hub of communication and also takes care of authorization. Distribution center for all mail/messages between services. Instead of services communicating with each other, they communicate with the gateway. Acts as the router of the MSA ecosystem. Helps with the abstraction of the msa ecosystem.

## Load Balancing

With services running in multiple nodes/servers/containers/deployment artifacts, you have a load balancer to balance outgoing traffic to instances of the service.

## Circuit breakers

Provides fallback so app doesn’t crash prevents cascading failures.

## Communication

### Http client

This is a class you can use for service to service communication in asp.net.

### Message queues

Research this. It’s a mechanism to communicate between services by registering to a service bus and sending messages that way (so its more secure).

## K8S (as a means of deployment)

Container orchestration, allows you to deploy multiple containers over a relatively private/encapsulated network.

# References:

[MSA Article from the interweb](https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63)