**Microservices**

**Date: 17-07-2021**

**Module - Microservices(MS) Fundamentals**

**Introduction**

* It is an architectural style that structures application as a collection of loosely linked services that implement business capabilities.
* Allows the company to evolve their technological stack

MS can be launched easily in a cloud-native environment compared to monolithic applications.

In MS, all communication is over REST.

Cost of MS:

* Single-threaded blocking calls. There are more but techademy is crap.

Techademy you suck.

Cloud-Native Microservice-Based

* Single code based
* Completely self-contained.

**Core concepts:**

**The services:**

All communication between services uses REST HTTP calls. Services operate on well-defined domains. Each service can consume any other service over the same communication strategy. (Mostly HTTP). There are some problems using this method though.

**Scale and Distribute:**

Each service is accessed is over the remote network call. That means services can be placed anywhere over the world and it would still work because of remote calls.

When a load on a particular service is increased, it can be scaled to new instances of nodes to handle the traffic.

**Latency:**

The communication pattern is costly due to remote calls. The risk of latency increases as more services are communicated for a need. Gridlock is possible.

Remedy being Circuit breaker pattern.

*Use a Circuit breaker:*

* Standard flow through your app
* When latency disrupts Timeouts start occurring, this trips the circuit and default behavior starts running.
* This may cause reduced functionality of your system but is better than a complete failure.
* The circuit closes when operations return back to normal.
* **Netflix** uses **histrix** as the implementation of the circuit breaker.

Strong timeouts,

* Global distribution
* Scaling of individual services
* Leveraging patterns like a circuit breaker are key takeaways.

**Bounded Context:**

Size of microservices, one common strategy is to leverage **Domain Driven Design** pattern, to decompose a large application into individual services.

* *Investigate the working system.*
* *Determine Domains.*
* *Break services accordingly.*

**Data Domain:**

**BASE Model:**

Strive for eventual consistency.

**API Layer:**

Protect the exact implementation of services from being viewed by clients/Outside world.