**Microservice:**

Agility

Scalability

Distribution

Do more with fewer resources

**Monolith:**

Single binary web artifact.

HTTP Message

-Presentation

-Business Process

-Data Access Layer

-Facade

-Processing

-I/O

Passthrough layers to satisfy architecture

Separation on concerns

Code decomposition into functional components

X High Coupling -makes changes hard - can still break downstream components due to high coupling - so need to test the whole artifact

X Higher complexity, higher coupling

X More time to build, test, deploy

Takes days to deploy and prod verification - high cost

Issue in packaging and code function. All processes in one package

Day 1 feature broken at later stage. Whole package suffers.

Performance takes a toll even when one process underperforms

EAR - Enterprise App Archive

Maintainability and Agility takes hit

**SOA : Service Based Architecture**

Broke our application into smaller module

1. Web technology was based on SOAP.

Makes compromises on how web worked

Every request is not OK 200 or Fault 500 .

Recoverable errors can occur

1. Aggregation layer - SOA Bus.

Transformation of XML and logic operation started getting added to SOA bus

Client ---- Service

Added new level of coupling

SOAP - Strong contract

WSDL - Web Services Description Language. Describes functionality offered by Web Service

Inherent Documentation Layer

Envelope - Wasted Space

Everything is Ok or Fault (200 - 500 ) No other

Deployment issue solved. Wire them services

Wiring became part of codebase. Management difficult

Expensive

**Microservice :**

Agile, Easily deployable on Cloud Native Env

Decompose the problem.

Modular, decoupled.

Building the services right size

ReST communication

Polyglot development possible

Each unit of work can be called from any other unit of work

Call any service

Cheap. Can be done with OpenSource Software

Agility and Distributability

High complexity - too many components to manage

Determining where all the code lives

A complicated deployment process - a difficult issue

Automate the deployment

Dramatic increease in network communication between compoennts, latency seen

A slow call can cause thread blocking

Reliability decreases with more part

One sick microservice can affect a lot

Cloud native - based on 12 factor methodology

The Twelve Factors

I. Codebase

One codebase tracked in revision control, many deploys

II. Dependencies

Explicitly declare and isolate dependencies

III. Config

Store config in the environment

IV. Backing services

Treat backing services as attached resources

V. Build, release, run

Strictly separate build and run stages

VI. Processes

Execute the app as one or more stateless processes

VII. Port binding

Export services via port binding

VIII. Concurrency

Scale out via the process model

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity

Keep development, staging, and production as similar as possible

XI. Logs

Treat logs as event streams

XII. Admin processes

Run admin/management tasks as one-off processes

To run in cloud infrastructure

Private, Public, Hybrid

Global distribution

Both are different - can make a monolithic that are cloud native

But they have merged

Cloud Native Microservice Based

* Single code base
* Completely set independent
* Zero file system usage. For that can migrate to  AmazonS3 or remote file system

**How Services Communicate**

* Rest over HTTP for all
* Services exposed via REST

No common rules- so has to be developed

Any language/framework that supports ReST over HTTP

Protocol Aware Heterogeneous Interoperability

* Services bound to a protocol
* Execute communicate over it

Expose services - other consumers need not have knowledge

* Agility
* Can deliver quickly
* Teams work in their own domain
* Expose the contract
* SOAP has its own contract

One Set of related function with little or no cross domain functions

Domain Driven Design

Provide low level data focus services - domain specific crud objects

Smaller services build test deploy faster

Any service can call any service

* Strong passivity rules needed
* With Versioning strategy

Distributed model

* Putting services all over the world - costly , high latency

Highly scalable.

Each service called by remote data call .

Each service can be individually scaled. No need to scale the whole component

In monolith, we build with busiest day. In Microservice - average day.

Costs-

Every single call has latency - as it is over the network

Response time increases under service load

Can cause gridlock

-Circular service call - 1---2 call each other

Add circuit breaker - trips the circuit when high latency.

Loses some functioality - with degraded performance - not whole gridlock.

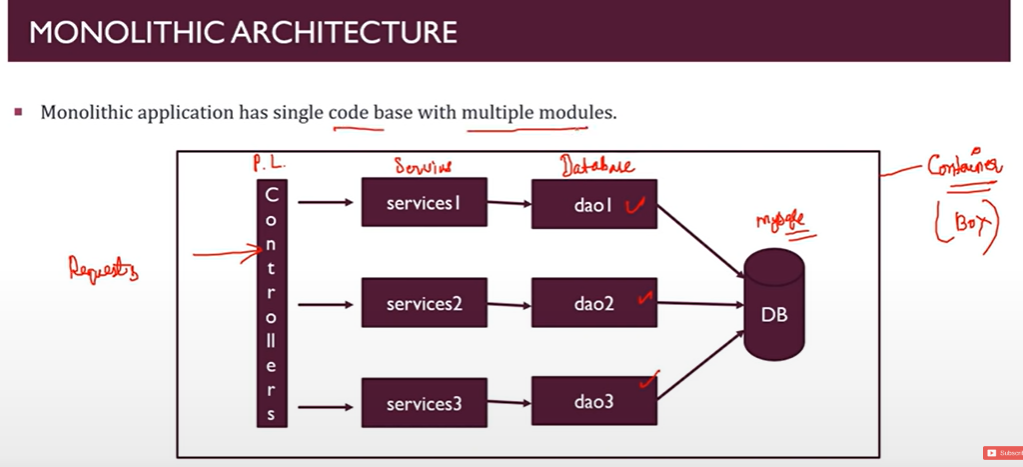
Netfilx offers Hysterix

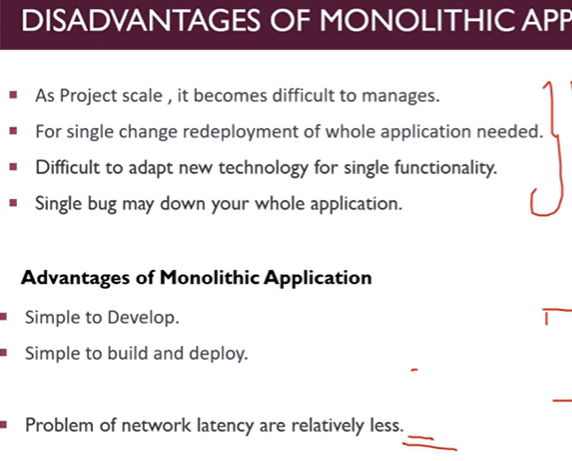
----------------------------------------------------

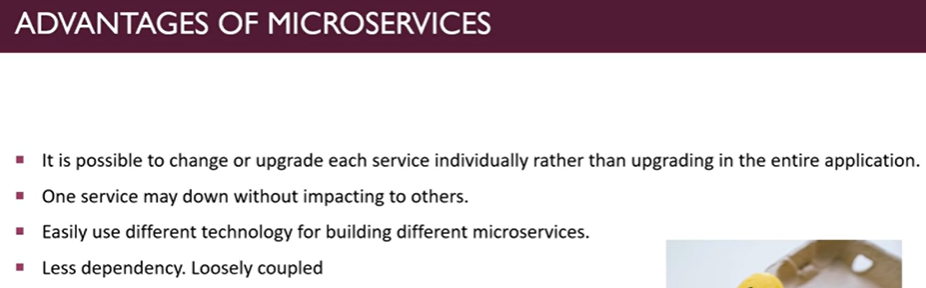
Monolith:

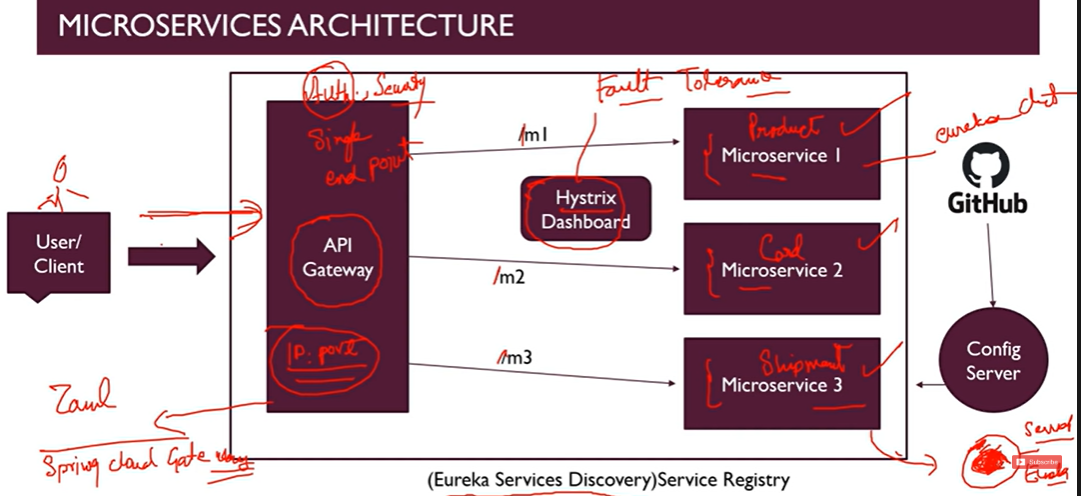
One single jar file deployed:

Under one project - single codebase









user-service  9001

contact-service 9002

api-gateway  8999

Msgs sent via restTemplate

Create a bean for it

Eureka Server to have Spring Cloud NEUreka Netfilx

@EnableEurekaServer

eureka.client.register-with-eureka=false = khud se mat segister ho jana

Add Discovery client dependency to client

Spring checks 8761 par server chal rha hai?? To spring ne configure kar diya

Service name - can be used anywhere in any service

Even though IP, host name changes

@LoadBalaced = add in restTemplate of eureka server to balace load

API Gateway MIcroservice

spring:

  application:

    name: api-gateway

  cloud:

    gateway:

      routes:

        - id: user-service

          uri: lb://user-service

          predicates:

            - Path=/user/\*\*

        - id: contact-service

          uri: lb://contact-service

          predicates:

            - Path=/contact/\*\*

Take me to that service when the particular URL is called

Gateway - spring cloud routing or JOule -

This is also a eureka client

can add actuator

Summary:

Make 2 microservices

Configure ports

Call each other data using restTemplate – make a bean, use it

Add Eureka Server

Register services there – EnableEurekaServer/Client

Prevent server registration

@LoadBalanced on Server

Add dependencies – refer project for that

Name the services – call the restTemplate using name

Make API gateway service- register

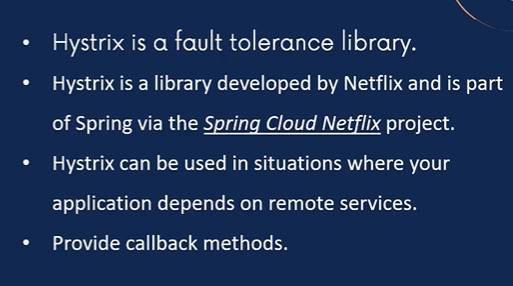
add which service to call at which path

---------------------------------------------------------------------

**Fault Tolerance**

MicroServices should be reliable

When high communication – fluctuation possible



Call back methods can be used in case of failure

Add the dependency

Configure – callback methods

Enable stream – to view in dashboard – Hysterix – realtime monitoring

YML takes 2 spaces for next level

Configure fallback service

@EnableHystrix

<dependency>

    <groupId>org.springframework.cloud</groupId>

    <artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>

    <version>2.1.3</version>

</dependency>

http://localhost:8999/actuator/hystrix.stream