Micro Service – It’s Architectural Pattern (It’s a way of developing enterprise application).

Monolith Arch vs Microservice Arch.

1. Functional Requirements –
2. Non-Functional Requirements (NFR) – Availability 24/7\*365 – 99.9% 99.99% Load Balancing, Fault-Tolerence, Logging, monitoring, performance

Service to Service Communication

1. Created a monolith (Bank app – Customer, Accounts, Transaction )
2. Micro service (Service Discover/ Service Registry, [Eureka/ Consul/Zoo Keeper], 3 diff micro-service (3 diff spring boot proj) ]
3. Back -end [Postman, API-Documentation (swagger) – OAS, cURL ]
4. DB – SQL based RDBMS (h2/MySQL/Postgres/Oracle)
5. Internal & External Dependencies (start.spring.io) – swagger, JWT
6. Docker – DevOPs tool – Containers
7. Sync vs Async Communication
8. Sync (RestTemplate – HttpClient - DiscoveryClient ) – FeignClient
9. Async (WebClient - WebFlux (reactive))
10. Packaging structure – Must have (model,controller, repo/dao, service), good to have (util,config,exception,dto,enums)
11. Application.properties / application.yml

Without API Gateway

http://localhost:8081/api/customers

http://localhost:8082/api/accounts

<http://localhost:8083/api/transaction>s

With API Gateway (8080) – API Gateway port number [Routing, Load Balancing, Centralized Authentication/Authorization]

http://localhost:8080/api/customers

http://localhost:8080/api/accounts

<http://localhost:8080/api/transaction>s

CORS – Cross Orgin Resource Sharing – Disabled locally

Centralised Logging – ELK Stack ( Elastic Search, LogStash, Kibana)

Monitoring – Prometheus & Grafana

Config Server – Github/Gitlab/BitBucket --- Repository (Locally)

Docker – Docker Images, Docker Container (Class & Object)

Docker Image – Is a blue print

Docker Container – Running instance of a docker image.

Docker Image = jar file + run time environment + config (dockerfile)

Javac 🡪 HelloWorld.java (Compilation Step) 🡺 HelloWorld.class

“java HelloWorld” – Invoking JVM to run HelloWorld class (Running HelloWorld class inside jvm)

“java -jar sts.jar”

“mvn spring-boot:run”

Single independent runnable unit – Which will run anywhere

* Enable WSL (Windows Subsystem for Linux) - $ -- Linux/ Unix environment
* Enable Hyper-V (Virtualization)

Git bash –

JVM – Java Virtual Machine –

Windows – Ubuntu

1. Dual boot (Install ubuntu along side windows)
2. Installing inside windows (Virtualization) – Heavy weight
3. Run with docker

# Containerization with Docker

## 1. Overview of Docker for Packaging Microservices

* **Introduction to Docker**: Container platform for developing, shipping, and running applications, ensuring app and environment consistency across development and production.
* **Microservices Packaging**: Each microservice is bundled with its environment and dependencies in a Docker container, removing host-specific issues (“it works on my machine”).

## 2. Benefits of Containerization for Microservices Deployment

* **Portability**: Run containers consistently across dev, test, or different clouds.
* **Resource Efficiency**: Lower overhead compared to VMs; containers start/stop faster.
* **Isolation & Security**: Each container is isolated, reducing attack surface and conflicts.
* **Scalability**: Easy to scale individual microservices horizontally.
* **Developer Productivity**: Each service can use its best-suited language/stack and be tested or deployed independently.
* **Service Discovery & Orchestration**: Containers enable service auto-discovery and orchestration with tools like Kubernetes.

## 3. Writing Dockerfiles and Building Containers for Microservices

* **Dockerfile Basics**: Instructions for building the image (FROM, WORKDIR, COPY, RUN, CMD).
* **Microservice Example**:

FROM python:3.9-slim

WORKDIR /app

COPY . .

RUN pip install -r requirements.txt

CMD ["python", "app.py"]

* **Directory Structure**: Each service has its source folder and Dockerfile; use docker-compose for orchestration.
* **Commands**:
  + docker build -t <my-service> .
  + docker run <my-service>

# Orchestration with Kubernetes

## 1. Introduction to Kubernetes: How It Works With Docker

* **Kubernetes Basics**: Container orchestration platform that manages container deployment, scaling, networking, and health checks.
* **Why Pair With Docker**: Docker handles building/running containers; Kubernetes manages them at scale in production.
* **Key Benefits**: Automated operations, service discovery, load balancing, self-healing, rolling updates, simplified management.

## 2. Kubernetes Concepts for Microservices

* **Pods**: Smallest deployable unit; contains 1+ containers.
* **Deployments**: Declarative updates for Pods, enabling rolling updates and rollbacks.
* **Services**: Abstracts a group of Pods to provide consistent network endpoint and load-balancing for internal traffic.
* **Ingress**: Handles external access to services (routes HTTP/HTTPS traffic to the appropriate service, SSL termination, URL-based routing); complements Kubernetes Services by consolidating external entry points.

## 3. Scaling and Managing Applications

* **Horizontal and Vertical Scaling**: Increase replicas or resources per service as needed—auto-scaling supported.
* **Self-healing Infrastructure**: Failed containers are restarted automatically.
* **Traffic Routing and Load Balancing**: Distributes requests evenly across healthy Pods.
* **Secrets and Config Management**: Separates configuration from code.

# Testing Microservices

## 1. Types of Testing for Microservices

* **Unit Testing**: Test individual service logic in isolation.
* **Integration Testing**: Validate service interactions with dependencies like databases or other services.
* **Contract Testing**: Verify that services adhere to interface contracts, e.g., with Pact, to prevent breaking APIs.
* **End-to-End (E2E) Testing**: Validate that the overall business workflows across multiple microservices work as intended.
* **Other Types**: Chaos testing (fault resilience), performance testing (load handling).

## 2. Best Practices for Robust Service Interactions and Deployments

* **Isolated Test Environments**: Mirrored to production for accurate test results.
* **Realistic Test Data**: Use representative, isolated test data.
* **Shift-Left Testing**: Integrate tests early in the CI pipeline.
* **Automated Testing & Observability**: Monitor with tracing/logging to quickly identify integration errors or slowdowns.
* **Service Independence**: Ensure each service is independently testable and deployable, encouraging faster releases and robust systems.
* **Collaboration**: Foster developer-tester-ops communication to resolve issues quickly.

# Continuous Integration & Deployment (CI/CD)

## 1. Implementing CI/CD Pipelines for Microservices

* **CI/CD Basics**: Automates build, test, deployment of code as changes are made.
* **Importance for Microservices**: Handles concurrency, ensures reliable, scalable delivery of many small, independent services.

## 2. Popular CI/CD Tools

* **Jenkins**: Flexible, open-source; best for complex, self-hosted pipelines but can be heavy for large microservices.
* **GitLab CI**: Integrated with GitLab; great for projects within the GitLab ecosystem.
* **CircleCI**: Cloud-native, simple setup; scalable and suitable for fast-moving microservice teams.

## 3. Deployments with Helm in Kubernetes

* **Why Helm**: Kubernetes package manager for versioning, templating, and managing complex, multi-service deployments.
* **Using Helm**:
  + Package services as Charts
  + Use helm install and helm upgrade to deploy/update applications with templates and version control.