**Monothilics Architecture:**

* Multiple components are combined in a single large app.
* Single code base
* Deployed in single bundle.
* change in one service then whole app is redeployed.
* one service down whole project down
* problem in scaling

**Microservice:**

* Devide large projects into smaller projects
* we buid each service seperately and these services communicates with each other with the help of Rest APIS
* Different codebase- we can code each service i different langiage like java,python,node js
* each module is managed independently->product,user,order,payment
* different tech stack
* Handling microservices is complex.
* can be suggested for large projects not for smaller.
* Each service can have its own database.

Microservices uses API gateways to get and pass the requests to the service.

Service Registry stores the details of each service. It is used to track all the information about services.

In Microservices each microservices can have different database, and each microservice can be developed, deployed and scaled independently

Each service can communicate with each other.

Communication can be synchronous or asynchronous.

Synchronous means we use HTTP apis to communicate.

Asynchronous measn we use message broker service to communicate with each other for ex- Apache kafka

Key components in Microservice architecture:

API gateway, Service Registry., Config Server, Distributed Tracing

How MicroServices architecture works:

Client sends request to API gateway then API Gateway sends the request to the particular service.

In Service Registry we register all the services so API gateway uses service registry to call the particular service.

Config server stores the configurations of the each microservice.

Distributed tracing stores the logs of all the services in the application

@**Controller**=> used to mark java class as controller

@**ResponseBody**=> Used to tell controller that object returned is automatically serialized into json and passed back into the HTTP Response Object

Instead of using these 2 annotations spring introduced new annotation called @**RestController** which combines both above annotations.

**@RequestParam**=> used to get the data passed in the request URL or API

**@RequestBody**=> responsible for retrieving the HTTP request body and automatically converting it to the java object. It internally used spring provided **http message convertor** to convert Json object into java object

**DTO-Data Transfer Object**

* These are used to transfer data/object between client and server
* Client sends it to DTO and then DTO sends it to server
* Server sends it to DTO and DTO sends it to Client
* DTO are introduced to overcome the security problems like if we have username, password in our application then it is not supposed to give these info to client in response to overcome this DTO are used.
* In DTO we add only the required fields that are supposed to transfer to user in response.

**ModelMapper:**

* It is a library which is used to convert Jpa entity into DTO and vice versa.
* To use this library, we have to add Model Mapper dependency into pom.xml file

**MapStruct:**

* It is also a library which is used to convert JPA entity into DTO and vice versa.

**API Documentation:**

* Springdoc-openai library is used to get the API documentation.

**API Gateway:**

**Service Registry**: Maintains the host names and port numbers of microservices. API gateway gets the hostname and port name from SR.

**Distributed Tracing:** IT helps us to find call hierarchy in microservices from start to end.

TO call one microservice from another we can do in different ways:

Rest Template, web client, spring cloud open feign.

**Circuit Breaker Pattern:**

If one microservice continuously calls another microservice and that microservice is down, then that failed microservice returns the error to the called microservice and then that first microservice returns that error to the API gateway and then API gateway send that error message to Client.

So, to avoid such issue circuit breaker pattern is implemented where the down microservice is configured with some default message and whenever it goes down then sends that default message to the client.

**Microservices Communication:**

Microservices communication can be created in below 3 ways:

1)RestTemplate=>not suggested => we create rest template bean in main class and then in service impl class we autowire it and then by using its object we call its getForEntity method and pass the Url and response class.

2)WebClient=> It is part of spring webflux dependency. We have to add webflux dependency in pom.xml file to use web client

3)Spring Cloud OpenFeign => for using this we have to add spring cloud dependency and open feign dependency in pom.xml file

Then we have to add @EnableFeignClient annotation on the top of main method class

Then we have to create interface and mark it as @FeignClient(url, value)

Then in that in/f we have to declare the other service method which we want to call like as below:

**public** **interface** ApiClient {

@GetMapping("/api/departments/getDepartment/{departmentCode}")

DepartmentDto getDepartment(@PathVariable String departmentCode);

}

**Service Registry:**

Spring Cloud Netflix Eureka is used to implement Service Discovery.

In SR we register all the service in our application and SR is used to track the status of all the MS which are UP, and which are down and track their instances

**TO Create Service Discovery below steps are needed:**

1). Create No rmal Service and add below dependency:

spring-cloud-starter-netflix-eureka-server

spring-cloud-dependencies

2). Then Disbale Eureka server as Eureka client by adding below lines in app.prop file.

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

3). In main () class add below annotation:

@EnableEurekaServer

@EnableEurekaServer=> Used to mark current project/service as Eureka Service Discovery

//below property is used to Unregister Eurek server as Eurek client

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

To mark any service as client of Service Discovery/Eureka Server we have to add Eureka Discovry Client dependency in client service.

@EnableEurekaClient=> Is used to mark particular service as client of service discovery

**To Register Particular Service into Service Discovry below steps are needed:**

1. Add Spring cloud and Spring Cureka client dependencies in pom.xml

spring-cloud-dependencies

spring-cloud-starter-netflix-eureka-client

1. In Application.properties file add below line:

spring.application.name=Department-Service

eureka.instance.client.serverUrl.defaultZone=http://localhost:8761/eureka/

1. In main() method class add below annotation:

@EnableDiscoveryClient

**How To create Multiple instances of MS:**

Create the jar file of application/Ms in STS or eclipse it will get created under target folder.

Thhen in Terminal run below command :

java -jar -Dserver.port=8082 target/department-service-0.0.1-SNAPSHOT.jar

**Load Balancing:**

**If there are multiple instances of each service and client call one instance of service but its down then automatically the another instances is called by eureka server which is Up is called as load balancing**

Spring provides Spring Cloud load balancer module to handle load balancing.

Load Balancing means creating multiple instances of service.

**API Gateway:**

It is a middleware between client and service.

When user sends request to it goes to API gateway where all services configurations is available and then API gateway routes that request to particular service.

**Main Purpose of using API gateway:**

1. Routing Request-Routing request to particular request.
2. Load Balancing- passing request to service which is Up and running
3. Security- Verifying the client before sending its request to service.

Spring Cloud Gateway dependency is used to create API Gateway

spring-cloud-starter-gateway.

Api gateway is a client for eureka server/service discovery so , we have to register API gateway service as client of Eureka server

By defauly API Gateway uses Netty Web Server.

AG uses the SR to discover port and services and routes that request to particular service.

We can configure API gateways in 2 ways:

1. Properties
2. Programatically

**eureka.instance.prefer-ip-address=true => This property is used to**

When this property is set to **true**, it means that the registered instance of your application prefers to be identified by its IP address rather than its hostname.

**Spring Cloud Config Server:**

**It is nothing but the spring boot project with spring cloud config server dependency.**

**Config server will be the client for service discovery, so we must add Eureka Client dependency in config server.**

**Advantages of using config server:**

1. **Whenever we change the config setting of microservice we don’t need to restart the service.**
2. **Externalize all the configuration of all the microservices in a centralized place.**

**Steps to Create Config Server:**

1. **Create spring boot project with below dependencies**

spring-boot-starter-actuator

spring-cloud-config-server

spring-cloud-starter-netflix-eureka-client

1. **Add below annotation on top of main() method class:**

@EnableConfigServer

1. **Make the config server as client of Eureka Server/Service Discovery by adding below property in app.properties file**

eureka.instance.client.serverUrl.defaultZone=http://localhost:8761/eureka/

1. Create the Git repo as config server and add its properties in app.prop file

spring.cloud.config.server.git.uri=https://github.com/shyamdhage000/microservices-config-server

spring.cloud.config.server.git.clone-on-start=true

spring.cloud.config.server.git.default-label=main

1. **Mark any service as client of config server by adding below dependencies in its pom.xml:**

spring-cloud-starter-config

1. Then add below lines in the app.properties file of that service which you want mark as config client

#Config Server Properties

spring.application.name=DEPARTMENT-SERVICE

spring.config.import=optional:configserver:http://localhost:8888

**Spring Cloud Bus:**

Spring Cloud Bus is a part of the Spring Cloud framework that provides a lightweight way to distribute messages across a Spring Boot application.

Spring Cloud Bus facilitates the propagation of configuration changes across multiple microservices. When a configuration change occurs, it can be broadcast to all connected microservices, ensuring that they update their configurations accordingly.

Spring Cloud Bus is commonly used with the Spring Cloud Config Server to trigger a refresh of the configuration properties in the microservices. This allows live updates to the configuration without requiring a restart of the application.

Spring Cloud Bus typically uses message brokers like RabbitMQ or Apache Kafka for communication between microservices. This enables asynchronous communication, which is beneficial for scalability and fault tolerance.

**Distributed Tracing with Spring Cloud Sleuth and Zipkin:**

Suppose we have multiple microservices in our application and they are communicating with each other like first MS calls second MS and second MS calls third and so on, so to record this call hierarchy we used Spring Cloud Sleuth and Zipkin.

In short it used to trace the chain of MS calls from Start to end. It is also used to trace why particular MS takes so much time to execute.

Distributed Tracing has two Ids Span Id and Trace Id.

So here these ids are attached to each request and span id is different for each service and Trace id is common for all the services.

**Zipkin**: Is used to see tracing information through UI

In Zipkin UI we can see all the information about tracing like controller Name, Controller method, server addresss, and Span Id and Trace Id.

Client start date server start date, time taken by server to complete the request

**Circuit Breaker Pattern:**  
The Circuit Breaker Pattern is a design pattern commonly used in microservices architecture to enhance the resilience and fault tolerance of a system. It helps prevent cascading failures and provides a way to gracefully handle and recover from faults in a distributed environment.

The Circuit Breaker Pattern aims to prevent a microservice from repeatedly trying to execute an operation that is likely to fail, thereby reducing the impact of failures and improving the overall system stability.

Techniques used to handle this issue:

1. Fallback method
2. Circuit Breaker
3. Retry
4. Rate Limiter

Circuit Breaker States:

1. Closed State- when there is no problem in services then it is closed state
2. Open State- if one MS is calling another MS which is down then it is called Open State CB
3. Half Open State- if one MS is calling another MS which is failed then after some amount of time it goes into Half Open state , then if response is received then goes into Closed state or again it goes into Open State

<http://localhost:8080/v3/api-docs>

**Docker Notes:**

Docker is a platform and tool designed to make it easier to create, deploy, and run applications using containers. Containers allow a developer to package up an application with all parts it needs, such as libraries and other dependencies, and ship it all out as one package. This ensures that the application runs consistently across various computing environments.

Key components of Docker include:

1. Docker Engine: The core technology that allows building and running containers. It consists of a server, a REST API, and a command-line interface.
2. Docker Image: A lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, and system tools.
3. Container: An instance of a Docker image. It runs the application in an isolated process with its own file system, networking, and isolated process space.
4. Docker Hub: It used to store the images and then pull those images from Hub to different environments like DEV, SIT, UAT, PROD

We create the Docker File and from that Docker file we build the docker image and then we deploy that image to docker hub.

**Dockerfile:**

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

**First we have to create docker file in springboot project with necessary steps**

**Then we have to build the jar file of the project**

**Command to create docker image from docker file:**

docker build -t springboot-docker-demo:0.1.RELEASE .

**Command to see all the created images in workspace:**

docker images

**Command to run the docker image in docker container:**

docker run -p 8080:8080 springboot-docker-demo

hostport:cont port image name

**command to see running docker container:**

docker ps

**command to push local image to docker hub:**

docker tag springboot-docker-demo shyamdhage/springboot-docker-demo:latest

desc=> img\_nm doc\_id/repository\_name:Tag id

docker images

docker push shyamdhage/springboot-docker-demo:latest

**command to pull docker hub image to local:**

docker pull shyamdhage/springboot-docker-demo:latest

If we want to communicate one docker container with another docker container then we have to create docker network, through docker network they can communicate with each other.

**Command to create docker network:**

Docker network create springboot-mysql-net

docker network ls => to see network info

**deploy image in network**

docker run --name mysqldb --network springboot-mysql-net -e MYSQL\_ROOT\_PASSWORD=root -e MYSQL\_DATABASE=employeedb -d mysql

**To use mysql bash use below commands:**

docker exec -it bash

mysql -u root -p

**Docker Compose:**

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define a multi-container application in a single file, usually named **docker-compose.yml**, and then use that file to spin up the entire application stack with a single command.

Here we have to create one .yml file and within that we have to specify all the services details.

First create docker-compose.yml file

Then execute below cmd:

**docker compose up**

The docker compose automatically does all the work of pulling the images and running it into container we just have to create .yml file and we have to just this command it will automatically does all the work for us. => docker compose up

**Apache Kafka:**  
Apache Kafka is an open-source distributed event streaming platform used for building real-time data pipelines and streaming applications. It was originally developed by LinkedIn and later open-sourced as part of the Apache Software Foundation. Kafka is designed to handle large volumes of data in a fault-tolerant and scalable manner.

Apache Kafka is a event streaming platform which acts as a intermediate between services.

Apache Kafka is a software platform which is based on a distributed streaming process.

It is a publish-subscribe messaging system which lets exchanging of data between applications, servers, and processors as well.

A publish-subscribe messaging system allows a sender to send/write the message and a receiver to read that message.

In Apache Kafka, a sender is known as a producer who publishes messages, and a receiver is known as a consumer who consumes that message by subscribing to it.

Apache Kafka works as a mediator between the source system and the target system.

Thus, the source system (producer) data is sent to the Apache Kafka, where it decouples the data, and the target system (consumer) consumes the data from Kafka.

Use case:

transportation: Driver rider notification, food delivery notification

**Apache Kafka has below four APIs:**

**Producer API:** A producer is the one which publishes or writes data to the topics within different partitions. Producers automatically know what data should be written to which partition and which broker.

**Consumer API:**

**Connector API:** This API executes the reusable producer and consumer APIs with the existing data systems or applications.

**Stream API:** This API allows an application to effectively transform the input streams to the output streams.

**Kafka Topic:** It is a Entity in Kafka with a name, it’s like a table in database. Producer uses Topic name to write to the topic. Consumer uses the Topic name to Consume/read the data from topic.We can have any number of topics. Each topic has unique name

It is nothing but category in Broker which categories the data or messages like, txt data,json data, xml data etc.

**Partition:** A topic is split into several parts which are known as the partitions of the topic.

The data content gets stored in the partitions within the topic. Therefore, while creating a topic, we need to specify the number of partitions (the number is arbitrary and can be changed later). Each message gets stored into partitions with an incremental id known as its Offset value.

**Offset: It** is a sequence of ids given to messages as they arrive in partition, once offset is assigned it is never changed

**Consumer Group:** It contains one or more consumers working together to process the messages.

Kafka has a cluster and is divided into different Brokers and these Brokers are managed by the Zookeeper.

It is fault tolerance it means if one Broker is down another broker can serve the request.

Kafka Broker: It is a Kafka server which acts as an intermediate between Producer and Consumer.

Producer sends msg to Broker and Consumer gets msg from broker.

Producer does not send the msg to consumer address rather it sends the msg to Broker address.

**Steps:**

Before starting kafka we have to start zookeeper:

Go into the kafka folder and open the cmd:

Command to start zookeeper:

.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties

Command to start kafka server in another cmd in kafka same path:

.\bin\windows\kafka-server-start.bat .\config\server.properties

Command to create topic in broker in third cmd:

.\bin\windows\kafka-topics.bat --create --topic topic-example --bootstrap-server localhost:9092

WRITE SOME EVENTS INTO THE TOPIC

.\bin\windows\kafka-console-producer.bat --topic topic\_demo --bootstrap-server localhost:9092

>hello world

>topic demo

READ THE EVENTS

.\bin\windows\kafka-console-consumer.bat --topic topic\_demo --from-beginning --bootstrap-server localhost:9092

**In Kafka Producer uses KafkaTemplate to send messages to kafka topic.**

**JsonSerilizer** =>It will convert user object into json and write to kafka topic

EventSource library is used to read real time data from source

Event Driven Architecture Microservices:

It is a software design pattern in which applicaations communicate with each other by sending or receiving events or messages.

EDA is also referred as asynchronous communication.

EDA is a loosely coupled.

Loosely coupled means all the services are independent of each other and they don’t have to know about each other’s.

If the requirement is to use some classes from one service into another service, then we have to add th e service details from which we need classes into pom.xml file of service in which we need those classes .

Like below:

<dependency>

<groupId>net.javaguides</groupId>

<artifactId>base-domains</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

We needed some classes from base-domains service into order service so we have added above details into pom.xml file of order service.

If there are multiple consumers wo are consuming the messages from same topic then we have to assign them in different group

**RabitMQ:**

RabbitMQ is an open-source message broker software that facilitates communication between different applications or components by acting as an intermediary for messaging. It is designed to support multiple messaging protocols, including Advanced Message Queuing Protocol (AMQP), Message Queuing Telemetry Transport (MQTT), and others.

**Message Queue:**

Messages are stored in queues until they are consumed by the intended recipients. This allows for asynchronous communication between different components of a system.

It is a buffer or storage in RabbitMQ broker to store the messages.

The messages are produced by the producer in Queue and read by consumer from Queue.

Once message is read and consumed by the consumer then it is removed from Queue

So, message can be only consumed once

**Exchange:**

It acts as a mediator between producer and Queue, producer sends msg to exchanger and then exchanger sends it to queue.

**Routing Key:**

The Routing key is a key which acts as a address key for the queue.

By using the key exchanger decides to which queue msg should be delivered.

**Binding**: It is a link between queue and exchange

Commands used to install rabbitmq through docker:

docker pull rabbitmq:management

Command to start rabbit mq:

docker run --rm -it -p 15672:15672 -p 5672:5672 rabbitmq:management

Rabit Mq Url:

Localhost:15672

RabbitMQ uses AMQP Advanced Messaging Queuing Protocol for messaging

RabbitMQ producer uses RabbitTemplate class to send message to exchange.

Springboot automatically configures RabbitTemplate for us we just have to inject and use it.

We can bind single exchange to multiple queues using routing keys

For adding RabbitMQ into springboot project we have to add RabbitMQ dependency in pom.xml file

**When to use Kafka, and when to use RabbitMQ?**

Well, both RabbitMQ and Apache Kafka are popular message brokers that can handle long-running tasks, but they have different design philosophies and use cases.

RabbitMQ is a traditional message broker that is optimized for reliability and ease of use. It supports multiple messaging protocols and provides features like message queuing, routing, and delivery guarantees. RabbitMQ is commonly used in enterprise environments for mission-critical applications that require high availability and fault tolerance.

Apache Kafka, on the other hand, is a distributed streaming platform that is optimized for scalability and high throughput. It is designed to handle large volumes of data in real time and supports features like event streaming, message replay, and distributed processing. Apache Kafka is commonly used for big data applications, IoT, and real-time analytics.

If your application requires high reliability and ease of use, RabbitMQ may be a better choice. If your application requires high scalability and real-time processing of large volumes of data, Apache Kafka may be a better choice.

It's also worth noting that there are other message brokers and streaming platforms available that may be better suited for your specific use case. It's important to evaluate your options carefully and choose the one that best meets your requirements.