Microservices

Before Microservice:-

1. Monolithic architect : multiple module/components are combine into single project.
2. Its based on only one language.
3. Have to deploy in single bungle
4. If we make change in single component then it will affect on other module also and have to re-deploy whole appln.

After Microservice:-

1. Large apps are divide into small part. REST API’s are use to communicate to each other.
2. Module have different code base(used diff language).
3. Each module managed independently.

How microservice communicate to each other:-

Through REST template and Feign Client

We are making 3 service .

1. Hotel service. ---------🡪having own DB1
2. User service ---------🡪having own DB2
3. Rating service ---------🡪having own DB3
4. Service Registry -------🡪 register all 3 services there.

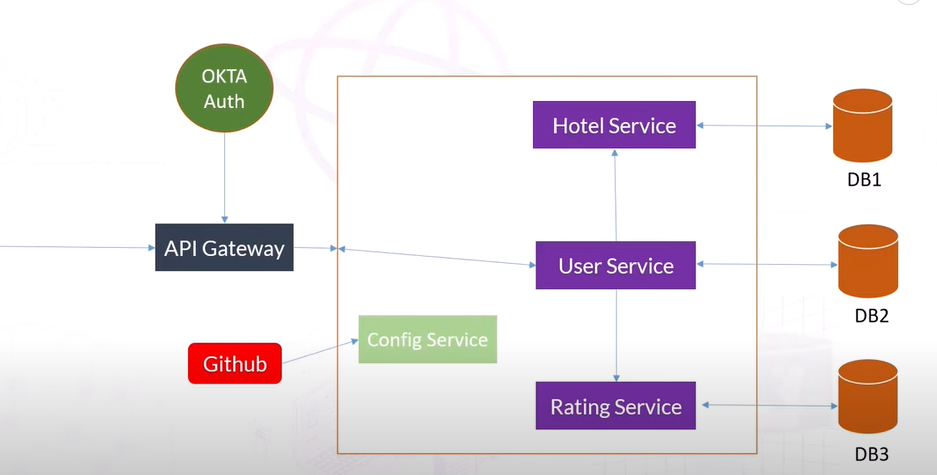
User service having user this will call to rating service and get to know how much rating user have given to particular hotel.

If this all services using same configuration then we are storing these on the config service/server.

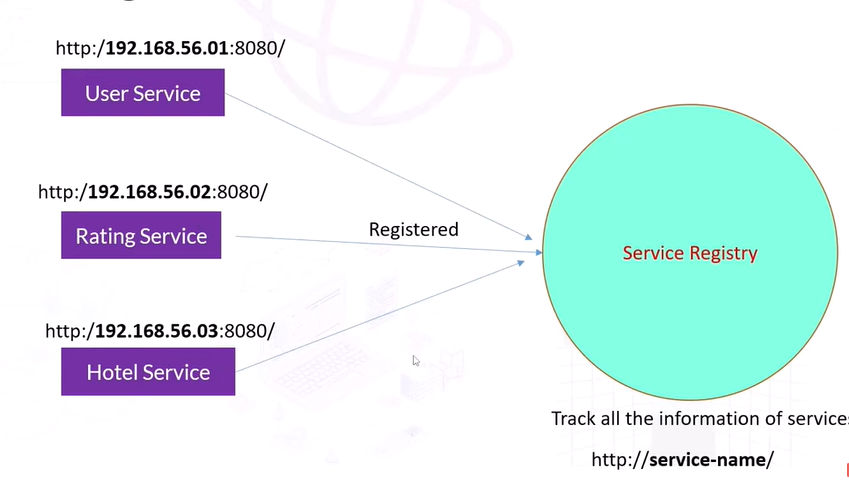
If any request is coming from client then it will go through API gateway to diff 3 servises.

OKTA Auth we are using for authentication its 3rd party service.

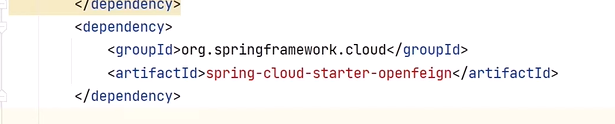
We are storing all services on Service Registry it will provide centralize registry .We can know which service is UP and Down.

****

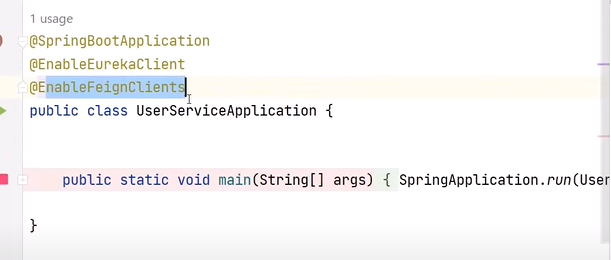
Service Registry:-



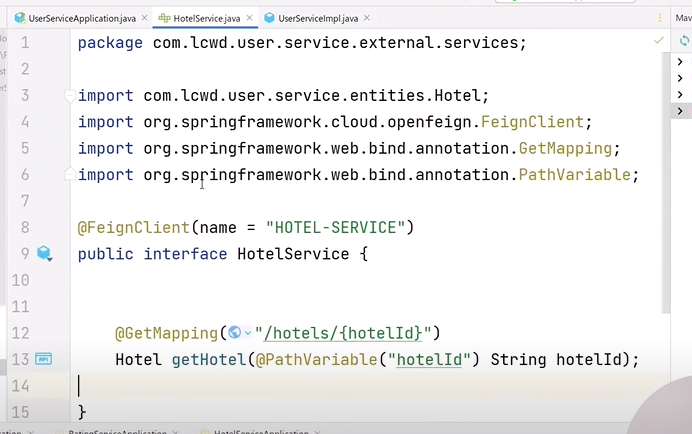


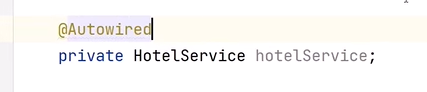


In springBoot mail application we are enabling Feign client



Creating a own interface with declaring the service name



Injecting the bean of this interface into impl class. 

In Impl class insteate of calling with REST template we are calling through feign client interface bean.



Working of Microservice.

RatingService

UserService

Request

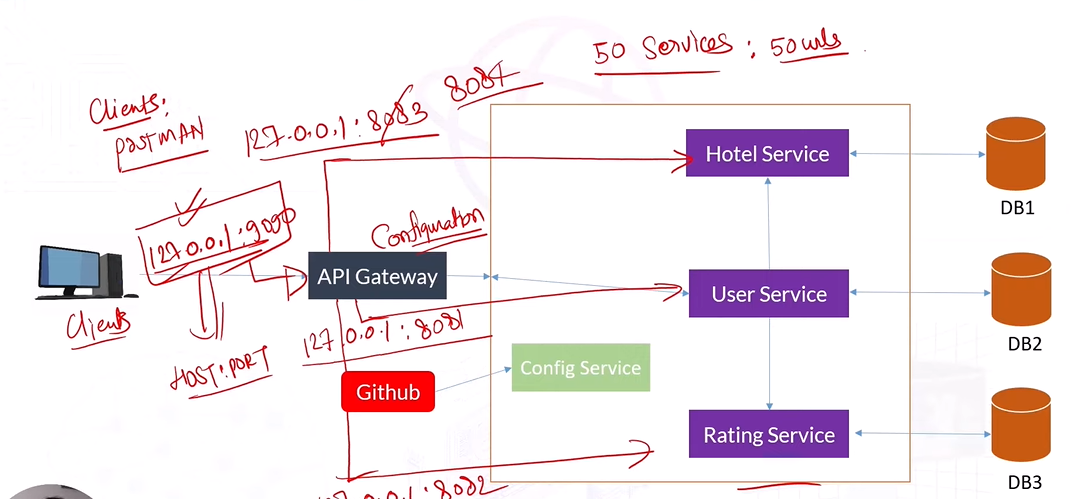
HotelService

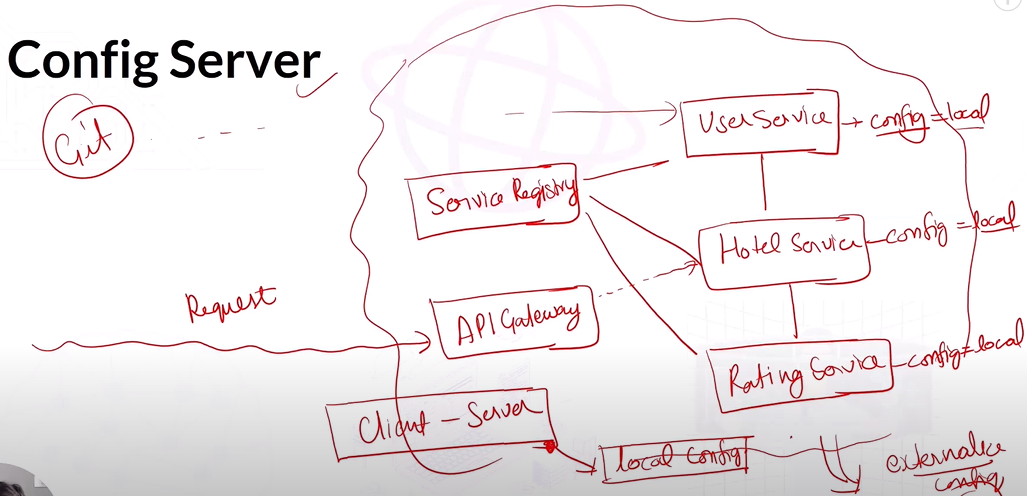
Response

**API gateway:-**

Client(postman) no need to call each service differently with there port no. through the API gateway we can communicate to each service. So client will call only API gateway and gateway can communicate to each other as per request.  
  
link for documentation:-

<https://docs.spring.io/spring-cloud-gateway/docs/3.1.9/reference/html/#glossary>

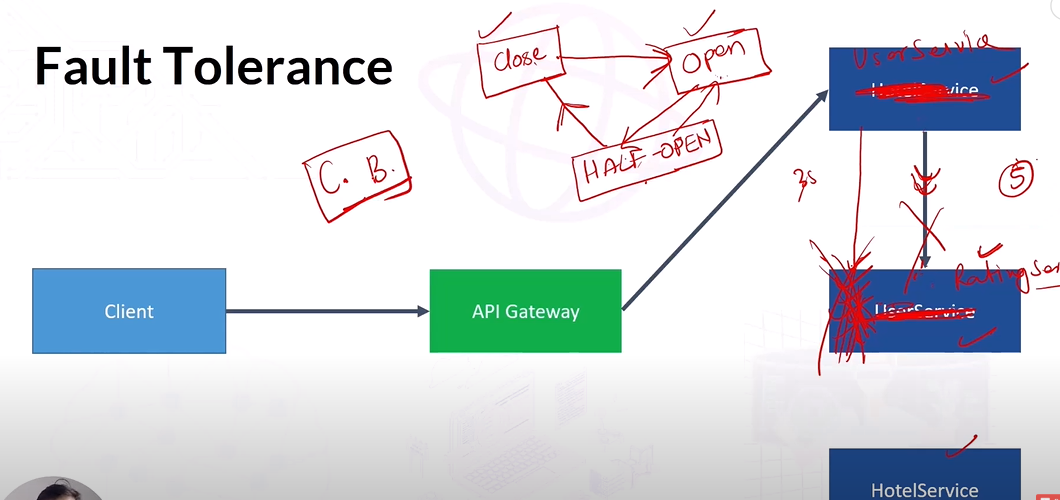




**Fault Tolerance:-**

If one service down while calling from different service then this is called as fault tolerance.

Will used the circuit breaker with min threshold value. With the help of circuit breaker will solve the fault tolerance issue by using resilence4j.



With the help of actuator we can monitor the health of application :-

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

With the help of AOP We can send the matrix to the actuator :-

The **spring-boot-starter-aop** dependency is used to enable **Aspect-Oriented Programming (AOP)** in a Spring Boot application. AOP is a programming paradigm that allows you to define cross-cutting concerns (e.g., logging, security, transaction management) in a modular and reusable way.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

For checking health:-

<http://localhost:8081/actuator/health>

Fault tolerance can be achive by  
1. Circuit breaker:-

A **circuit breaker** in Spring Boot is a design pattern used to provide fault tolerance in microservices. It prevents the system from making repeated calls to a failing service, protecting the application from cascading failures and degraded performance. Spring Boot integrates with **Resilience4j**, a popular library for implementing circuit breakers.

#### ****1. Add Dependencies****

Include the Resilience4j dependency in your pom.xml (for Maven):

xml

Copy code

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot3</artifactId>

</dependency>

A **circuit breaker** in Spring Boot is a design pattern used to provide fault tolerance in microservices. It prevents the system from making repeated calls to a failing service, protecting the application from cascading failures and degraded performance. Spring Boot integrates with **Resilience4j**, a popular library for implementing circuit breakers.

### ****Steps to Implement Circuit Breaker in Spring Boot****

#### ****1. Add Dependencies****

Include the Resilience4j dependency in your pom.xml (for Maven):

<!--

https://mvnrepository.com/artifact/io.github.resilience4j/resilience4j-spring-boot2 -->

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot2</artifactId>

</dependency>

#### ****2. Enable Circuit Breaker in Spring Boot****

Add the following annotation to your main application class:

java

Copy code

@SpringBootApplication

@EnableCircuitBreaker

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

#### ****3. Configure Circuit Breaker****

Define your circuit breaker settings in the application.yml file:

#-----------we are doing yml based configuration for circut breaker-----------#

management:

health:

circuitbreakers:

enabled: true

endpoints:

web:

exposure:

include: health

endpoint:

health:

show-details: always

resilience4j:

circuitbreaker:

instances:

ratingHotelBreaker:

register-health-indicator: true

event-consumer-buffer-size: 20

failure-rate-threshold: 50

minimum-number-of-calls: 3

automatic-transition-from-open-to-half-open-enabled: true

wait-duration-in-open-state: 5s

permitted-number-of-calls-in-half-open-state: 3

sliding-window-size: 10

sliding-window-type: count-based

#### **4. Use Circuit Breaker in controller**

/\*\*

\* this api is calling two service in UserServiceimpl class so we are using circuit breaker on this api

\* for this method ratingHotelFallBack the return type must be same

\* \*/

//get single user

@GetMapping("/{userId}")

@CircuitBreaker(name = "ratingHotelBreaker", fallbackMethod = "ratingHotelFallBack")

**public** ResponseEntity<User> getSingleUser(@PathVariable("userId") String userId){

User user = userService.getUser(userId);

**return** ResponseEntity.*status*(HttpStatus.***OK***).body(user);

}

//creating fall back method for ciruit breaker if any service down then only this method will execute

**public** ResponseEntity<User> ratingHotelFallBack(String userId, Exception ex){

logger.info("fall back is executed bcz service is down : {}",ex.getMessage());

User user = User.*builder*()

.userEmail("dummy@gmail.com")

.userName("dummy")

.aboutUser("this user is created bcz some services are down")

.userId("123")

.build();

**return** **new** ResponseEntity<User>(user,HttpStatus.***OK***);

}

#### ****5. Monitoring the Circuit Breaker****

You can monitor the circuit breaker health using Spring Boot Actuator. Add the following to application.yml:

yaml

Copy code

management:

endpoints:

web:

exposure:

include: health

health:

circuitbreakers:

enabled: true

Access the circuit breaker health status at:  
http://localhost:8081/actuator/health

2. Retry module

3. Rate limiter

1.(@RequestBody User user) with this annotation JSON obj data will convert into User obj.

@PostMapping("/create")

**public** ResponseEntity<User> saveUser(@RequestBody User user) {

User saveUser = userService.saveUser(user);

**return** ResponseEntity.*status*(HttpStatus.***CREATED***).body(saveUser);

}

[2.@PathVarialbe](mailto:2.@PathVarialbe) will take single parameter

@GetMapping("/{userId}")

**public** ResponseEntity<User> getSingleUser(@PathVariable String userId){

User user = userService.getUser(userId);

**return** ResponseEntity.*status*(HttpStatus.***OK***).body(user);

}

3. @Transient

**private** List<Rating> rating = **new** ArrayList<Rating>();

with the help of @Transient anno we are not storing in to Database

**ModelMapper** is a powerful Java library used for simplifying the process of mapping data between objects, particularly between **DTOs (Data Transfer Objects)** and **entities** in applications, including those built with **Spring Boot**. It helps eliminate boilerplate code that would otherwise be necessary for copying values from one object to another, making the code cleaner, easier to maintain, and reducing the risk of errors.

**Key Features of ModelMapper:**

1. **Automatic Mapping**: ModelMapper can automatically map fields between objects if the source and destination objects have matching property names.
2. **Flexible Configuration**: You can customize mapping behavior for complex cases where automatic mapping isn't enough, using property maps or custom converters.
3. **Supports Nested Mapping**: You can map nested objects automatically, reducing the need for manual mapping.
4. **Mapping Collections**: ModelMapper can handle lists or other collections of objects, allowing you to map them all in a single step.
5. **Type Safety**: It offers compile-time checks, ensuring the type safety of the mapping.

**Common Use Cases:**

1. **DTO to Entity Mapping**: ModelMapper simplifies the conversion of data between DTOs and database entities, making the code more readable and maintainable.
2. **Simplifying Object Transformation**: Instead of manually copying properties from one object to another, ModelMapper provides a clean and efficient way to transfer data.
3. **Handling Nested Objects**: ModelMapper allows mapping of nested objects (e.g., mapping a UserDTO to a UserEntity that may have a nested Address object).
4. **public** **class** Example {
5. **public** **static** **void** main(String[] args) {
6. ModelMapper modelMapper = **new** ModelMapper();
7. // Source object
8. UserDTO userDTO = **new** UserDTO("John", "Doe", 30);
10. // Map DTO to Entity
11. UserEntity userEntity = modelMapper.map(userDTO, UserEntity.**class**);
13. System.***out***.println(userEntity.getFirstName()); // John
14. }
15. }
16. **class** UserDTO {
17. **private** String firstName;
18. **private** String lastName;
19. **private** **int** age;
20. // Constructors, getters, setters
21. }
22. **class** UserEntity {
23. **private** String firstName;
24. **private** String lastName;
25. **private** **int** age;
26. // Constructors, getters, setters
27. }

Service Registery

-3 microservice are register with this ,load balancing

@EnableEurekaServer

By the below 2 imp property its will not become a own registery

It will not self register

client:

register-with-eureka: false

fetch-registry: false

Steps to deploy on GIT.

1. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git init
2. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git status
3. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git add UserService or git add .
4. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git status
5. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git commit -m "Deleted zip file"
6. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git config --global user.email "bhagyeshwadhale1998@gmail.com"
7. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git config --global user.name "bhagyesh"
8. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git config --global user.name bhagyesh
9. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git config --global user.email bhagyeshwadhale1998@gmail.com
10. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git commit -m "Deleted zip file"

[master (root-commit) 302fd1e] Deleted zip file

11.C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git branch

\* master

1. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git branch -M main
2. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git remote add origin <https://github.com/Bhagyesh108/microservice.git>
3. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git remote remove origin
4. C:\Users\ER. Bhagyesh\Desktop\microservice\_project>git push -u origin main
5. $ git config --list
6. ER. Bhagyesh@MSI MINGW64 ~/desktop
7. $ git config user.name
8. bhagyesh
9. ER. Bhagyesh@MSI MINGW64 ~/desktop
10. $ git config user.email
11. bhagyeshwadhale1998@gmail.com

**Steps for GIT:-**

ER. Bhagyesh@MSI MINGW64 ~/Desktop/microservice\_project (main)

$ git status

ER. Bhagyesh@MSI MINGW64 ~/Desktop/microservice\_project (main)

$ git init

Reinitialized existing Git repository in C:/Users/ER. Bhagyesh/Desktop/microservice\_project/.git/

**To adding all files in staging area**

ER. Bhagyesh@MSI MINGW64 ~/Desktop/microservice\_project (main)

$ git add --a

ER. Bhagyesh@MSI MINGW64 ~/Desktop/microservice\_project (main)

$ git add hotelservice

ER. Bhagyesh@MSI MINGW64 ~/Desktop/microservice\_project (main)

$ git commit -m "This is a hotelservice which store all info related to hotels"

**Git log to see all our commit**

ER. Bhagyesh@MSI MINGW64 ~/Desktop/microservice\_project (main)

$ git log

commit e285a627462f35f3c2897869481b6b75d9fe53b5 (**HEAD** -> **main**)

Author: bhagyesh <bhagyeshwadhale1998@gmail.com>

Date: Sat Dec 28 09:54:21 2024 +0530

This is a hotelservice which store all info related to hotels

commit 302fd1ea5b5e4f644860d80c3ba1345ce5c2c2ae (**origin/main**)

Author: bhagyesh <bhagyeshwadhale1998@gmail.com>

Date: Thu Dec 26 11:39:47 2024 +0530

**We are pushing changes into main branch**

ER. Bhagyesh@MSI MINGW64 ~/Desktop/microservice\_project (main)

$ git push origin main

Enumerating objects: 46, done.

Counting objects: 100% (46/46), done.

Delta compression using up to 8 threads

Compressing objects: 100% (28/28), done.

Writing objects: 100% (45/45), 12.64 KiB | 1.40 MiB/s, done.

Total 45 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)

remote: Resolving deltas: 100% (1/1), done.

To https://github.com/Bhagyesh108/microservice.git

302fd1e..e285a62 main -> main

ER. Bhagyesh@MSI MINGW64 ~/Desktop/microservice\_project (main)

$ gitER. Bhagyesh@MSI MINGW64 /E/softwares/learnwithdurgesh/enduser (main)

$ git remote -v

kafka-related Bhagyesh108@github.com:https://github.com/Bhagyesh108/kafka-rela

ted (fetch)

kafka-related Bhagyesh108@github.com:https://github.com/Bhagyesh108/kafka-rela

ted (push)

origin https://github.com/Bhagyesh108/kafka-related.git (fetch)

origin https://github.com/Bhagyesh108/kafka-related.git (push)

log

commit e285a627462f35f3c2897869481b6b75d9fe53b5 (**HEAD** -> **main**, **origin/main**)

Author: bhagyesh <bhagyeshwadhale1998@gmail.com>

Date: Sat Dec 28 09:54:21 2024 +0530

This is a hotelservice which store all info related to hotels

commit 302fd1ea5b5e4f644860d80c3ba1345ce5c2c2ae

Author: bhagyesh <bhagyeshwadhale1998@gmail.com>

Date: Thu Dec 26 11:39:47 2024 +0530

Deleted zip file

If remote origin is not set then

ER. Bhagyesh@MSI MINGW64 /E/softwares/learnwithdurgesh/enduser (main)

$ git remote set-url origin https://github.com/Bhagyesh108/kafka-related.git

ER. Bhagyesh@MSI MINGW64 /E/softwares/learnwithdurgesh/enduser (main)

$ git remote -v

kafka-related Bhagyesh108@github.com:https://github.com/Bhagyesh108/kafka-rela

ted (fetch)

kafka-related Bhagyesh108@github.com:https://github.com/Bhagyesh108/kafka-rela

ted (push)

origin https://github.com/Bhagyesh108/kafka-related.git (fetch)

origin https://github.com/Bhagyesh108/kafka-related.git (push)

ER. Bhagyesh@MSI MINGW64 /E/softwares/learnwithdurgesh/enduser (main)

$ git push -u origin main

Enumerating objects: 25, done.

Counting objects: 100% (25/25), done.

Delta compression using up to 8 threads

Compressing objects: 100% (18/18), done.

Writing objects: 100% (25/25), 9.03 KiB | 1.81 MiB/s, done.

Total 25 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)

remote: Resolving deltas: 100% (1/1), done.

To https://github.com/Bhagyesh108/kafka-related.git

\* [new branch] main -> main

branch 'main' set up to track 'origin/main'.

**It will delete .git folder iw will delete git repositiry**

ER. Bhagyesh@MSI MINGW64 ~/Desktop/microservice\_project (main)

$ rm –rf .git