================ Project Doubts Microservices\_basic\_Projects ===============

Student / Address Microservices project chesina mistakes and vachhina doubts

1st point : Microservices lo ye module ki a module separete Project and DB configuration cheyali.

----------------------- ------------- - Address project --------------------------- ----------

Address project lo ye doubts raledu but follw avvalsina key points

Point 1:

Best practies: kosam Rest End point ki Request and responces ni dirent ga Entity class ni use chesi kakunda Separt ga Request and responces ki java class ni create chesukovali

Point 2:

Addservices\_IMP class no noraml ga Rest end point ki retun chesetappudu direct ga Entity class ni eistham because prious ga ala chesevallam

But response class ni use chestam..

But …….. First Enity Class ni use chese db lo data ni store cheyagalamu and A Entity class nuodi mana ki kavalsina data ni responces class ki pass chestham

EX :

@Override  
public Addresponces addAddress(Addrequest addrequest) {

// Convert Addrequest DTO to Address entity  
 Address address = new Address();  
 address.setCity(addrequest.getCITY());  
 address.setState(addrequest.getState());  
 // Save the address entity to the database  
 Address savedAddress = addrepo.save(address);  
  
 // Create and return Addresponces object using the constructor  
 return new Addresponces(savedAddress.getAddid(), savedAddress.getCity(), savedAddress.getState());  
}

Use : Eila responces class ni use cheyatam valana Responces ga (postman ) lo manaki ye tags kavalo.. vati varike easy ga display cheyachhu

Point 3 :

Rest End point lo @PathVariable ni use chesthe Rest point ni { } lo ne evvali

EX: @GetMapping("/get/{addid}")

-------------------------------- Student Project -----------------------------------------------------

Main ga issue face chesindi E project lone

Point 1 : Normal ga rase code kakunda Extra ga E class ni rayali

@FeignClient(url = "${address.service.url}", value = "address-feign-client",  
 path = "/address")  
**public interface AddressFeignClient {**  
  
 @GetMapping("/get/{id}") // Note:- Address project lo ye rest end point echhamo same ade name tho eikkada evvali  
  **public AddressResponse getById(@PathVariable long id);  
  
}**

**Key points in AddressFeignClient class : Address Controll class lo ye rest end point vadamo same rest end point AddressFeignClient class lo kuda use cheyali**

**EX :**

--- Address Controller class

@GetMapping("/get/{addid}")

public Optional<Addresponces> getAddressById(@PathVariable long addid) {

return addService\_i.getAddressById(addid);

}

--- AddressFeignClient Interface : Address project and Student project Communicate avvali ane @FeignClient ni use cheyali

**@FeignClient(url = "${address.service.url}", value = "address-feign-client",  
 path = "/address") //Address project URL Evvali**public interface AddressFeignClient {  
  
 @GetMapping("/get/{id}") // Note:- Address project lo ye rest end point echhamo same ade name tho eikkada evvali  
 public AddressResponse getById(@PathVariable long id);  
  
}

**application.properties**

server.port=8082  
address.service.url=http://localhost:8080 // E URL ne “**AddressFeignClient “ Interface lo use chestham Edi Address project ki connect avvali ani Student Project ki chepthunattu**

**Point 2 :**

**Post Man lo Student Rest End point ni Register Chesthunnappudu Address out put tag’s null vacchy**

**Class :**

**public class AddressResponse {**

**@JsonProperty("addid")**

**private long AddressId;**

**@JsonProperty("state\_name")**

**private String state;**

**private String city;**

**}**

**Reason :**

**AddressResponse ni manam Student Project lo use chesamu Ala use chesinappudu**

**Student project nuodi Address details ni get cheyataniki   
Connect Avve URL : GET <http://localhost:8080/address/get/{id}> // Student URL ni use chesi details register chesetappudu manam pass chesian AddressID anedi E URL dwara Address Project ki connect avvi details ni get chesthundi**

**Now why tags passed null**

**Pyna cheppinattu pass chesina Address id ni base chesukoni Address details vasthayy kanuka manam Address Entity lo veriable name yela avithe rasthamo same Student Project loni AddressResponse class lo kuda alane rayali lekunte responces Null vasthundi.**

**Kavali ante @JsonProperty("state\_name") anotation ni use chesi manam coustom veribel name ni rasukovachhu gani coustom tags ni Post man lo thechhukolenu**

**-- coustom veriable names**

**Oka vela Student project lo ni AddressResponse class lo veriables name different ga evvali anukunte**

**@JsonProperty("state") // eikakda Address Entity class veriable ni evvali**

**private String state\_name;// eikkada manaki nachhindi evachhu**

**--- post man coustom tags “ out put “ lo**

**No, out put responces ki coustom tags evvatam kudaradu Entity class lo yela avithe eisthamo ave vasthayy..**

==================== **Project With Spring Cloud Eureka** ====================

First **Eureka** server ki oka Project Ni create chesukovali a Project lo Eureka server dependency ni add cheyali

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>**spring-cloud-starter-netflix-eureka-server**</artifactId>  
</dependency>

@EnableEurekaServer --> Anotation ni Main Springboot Application lo ryali.

E **Eureka** Server lo Anni Microservices Projects anevi rigister cheyali ala cheyataniki Every Microservices Projects application.properties file lo Eureka server URL ni Configure Cheyali

Ex : eureka.client.service-url.defaultZone=http://localhost:8761/eureka

Previous Ga Address and Student Project communicate avvataniki FeignClient Interface Ni create chesanu and Address ki related URL ni Student Project application.properties file lo rasi danni FeignClient Interface ki MAP Chesamu

But Spring Cloud **Eureka** vadithe application.properties file lo Hard Code cheyakunda service name tho communicate cheyachhu

EX :

spring.application.name=**Student-microservice-with-eureka**

eureka.client.service-url.defaultZone=http://localhost:8761/eureka // edi Project Eureka server lo rigister avvadaniki

@FeignClient(value = "address-microservice-with-eureka",  
 path = "/address") // Note:- application.properties file loni service name edi

public interface AddressFeignClient {  
  
 @GetMapping("/get/{id}") // Note:- Address project lo ye rest end point echhamo same ade name tho eikkada evvali  
 public AddressResponse getById(@PathVariable long id);  
  
}

**Using Eureka in Address and Student Modules project**

**Address Project lo application.properties file lo below tags rayali…**

spring.application.name=**address-loadbalancer-project // services name**

eureka.client.service-url.defaultZone = <http://localhost:8761/eureka> // Project **Eureka lo register Avvataniki**

**Student Project lo application.properties file lo below tags rayali…**

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

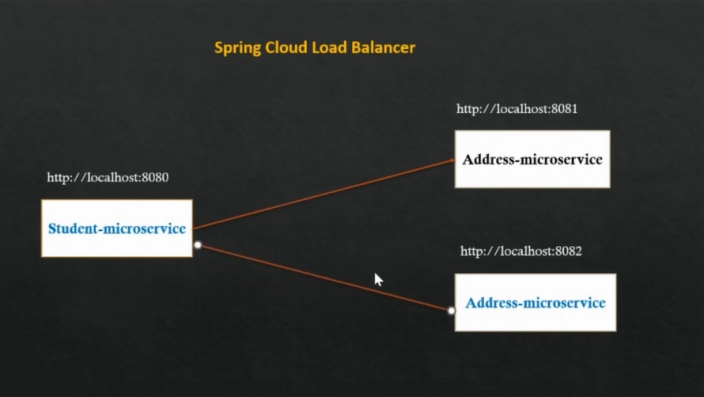
spring.application.name=Student-microservice-with-eureka

@FeignClient(value = "address-microservice-with-eureka",  
 path = "/address") // Note:-Value anedi Address project lo ni service name evvali  
public interface AddressFeignClient {  
  
 @GetMapping("/get/{id}") // Note:- Address project lo ye rest end point echhamo same ade name tho eikkada evvali  
 public AddressResponse getById(@PathVariable long id);  
  
}

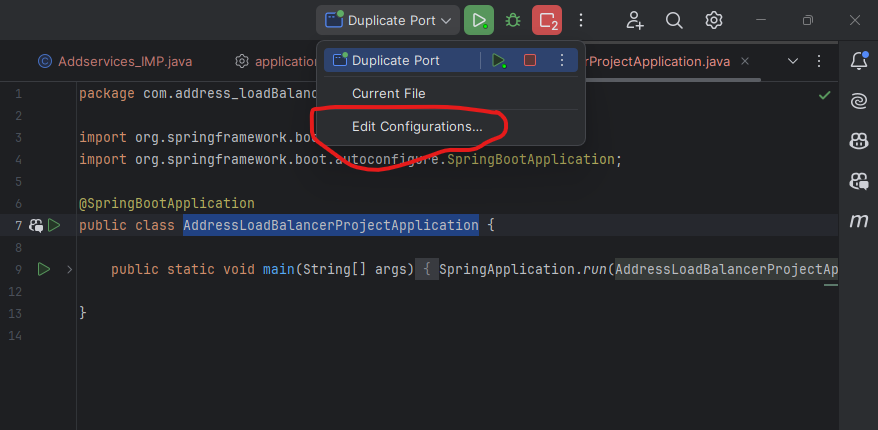
**===================== Spring Cloud LoadBalancer =====================**

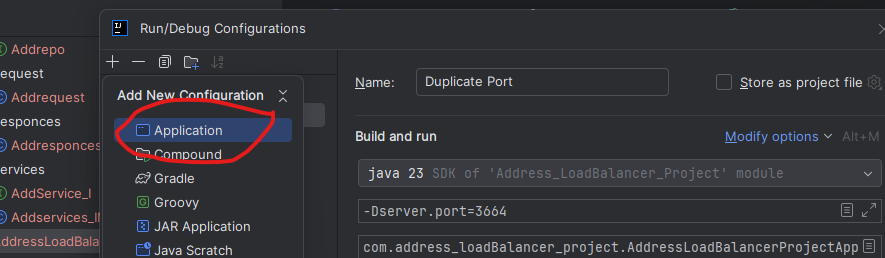
Oka microservice nuodi einko microservice communicate avvetappudu oka vela yekkuva request anevi vasthe Application Performances anedi thaggipothundi andukane .. veroka instance ni create chesi E LoadBalancer ni use chesthe Traffic load anedi equal ga spread avuthundi

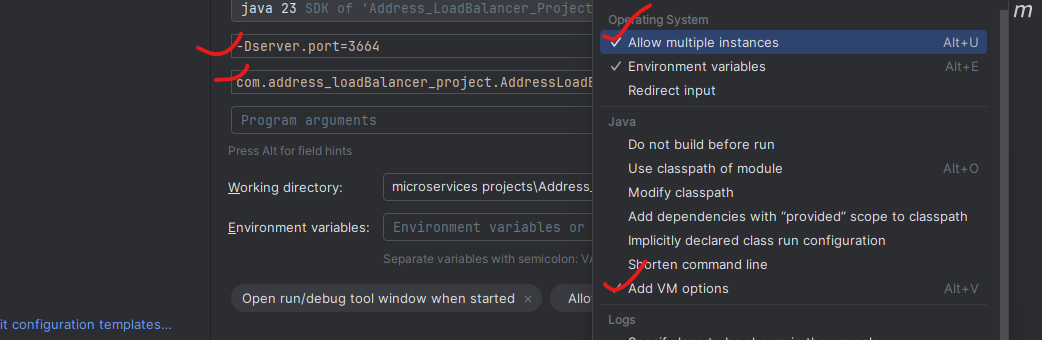
**Conclustion :** Same project ni 2 port numbers vadi run cheyatam, ala cheyatam valla project anedi 2 ports lo run avuthundi .. so request anedi renditiki velthundi

C 

**How to run two different port in intellij idea?**





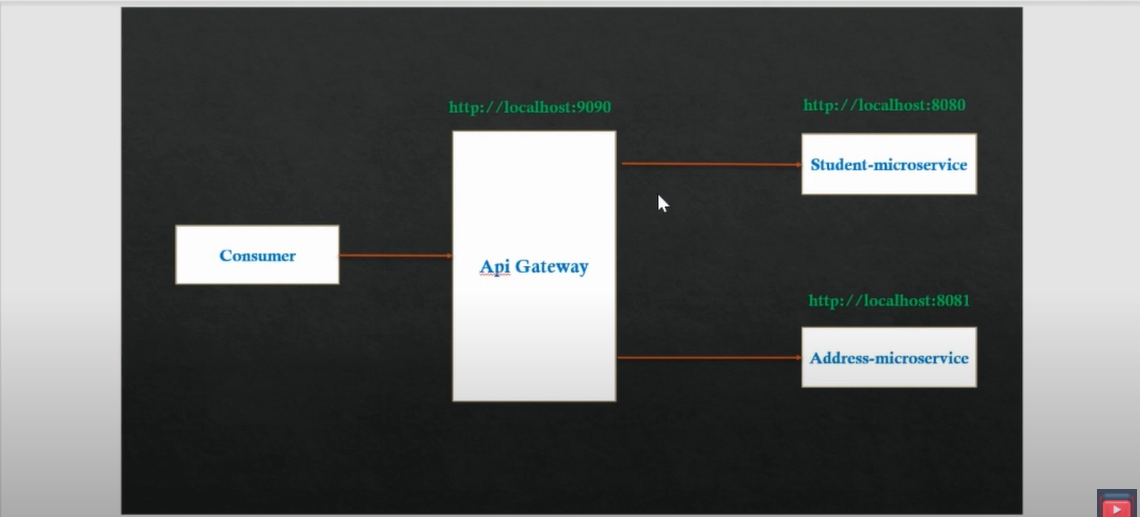


Eila 2 ports run chesinappudu 2 instances anevi Euraka lo create avvi load balences anedi adjust avuthundi.

**===================== Spring Cloud API GateWay =======================**

Consumer nuodi requiest vacchinappudu direct ga microservices project URL tho connect avvakunda. API GateWay anedi Middle ware ga work avuthundi

**Conclustion :** consumer request ki Microservices project ki middle lo ok conncection bridge la work avuthundi**.**



**First Oka API GteWay Project ni create cheyali andulo mana custome code rasukovachhu..**

Every Spring Boot Microservices Project Lo API GateWay ki connect avvataniki application.properties file lo rayali

**#It is used to access all the microservices via Api-gateway**  
eureka.instance.hostname=localhost  
  
eureka.instance.prefer-ip-address=true

API GteWay Project lo oka Coustom Filter create chesamu dani explanation because of **post-filter and pre-filter right ?**

coustom filter lo only ("/api/student") ye vundi remain Header details levu yendukani?

Ans :

Asalu e Coustom fileter leka poeina spring boot Application routing jaruguthundi ye Error vundadu

But eikkda just normal ga e ("/api/student") header ni echhi e url hit avinappudu some conditions work avvali ane danikosam ala rayachhu ani just example but as of now Ala ye conditions rayaledu just LOG ni print chesamu anthe

As per your explanation i didnt write the coustom filter class also it will works with out any error right  
this coustom filter is only for Checking credential right  
for that we can use Spring Securtity right ?

Exactly — you're right again. Let’s break it down:

## ✅ ****What is recommended in real-time projects?****

### 🔐 ****Use Spring Security for:****

Authentication (JWT, OAuth2, Basic Auth, etc.)

Authorization (role-based access)

Token validation

Securing routes

Handling 401/403 errors properly

Protecting APIs with filters in the correct security filter chain

✅ **Spring Security is the production-grade solution** for all security-related concerns.

## 🛑 ****When NOT to use**** GlobalFilter ****or**** CustomFilter ****for security:****

### GlobalFilter (from Spring Cloud Gateway) is:

Just a **request/response interceptor**

Good for **logging**, **tracing**, **custom headers**, or **rate limiting**

**NOT** designed for handling authentication, user sessions, or role-based access

### ❌ ****Not recommended**** for:

Checking JWT tokens manually

Controlling access based on roles

Redirecting or denying users

## ✅ So in summary:

| **Concern** | **Use** |
| --- | --- |
| Logging requests/responses | GlobalFilter is OK |
| Adding headers (like traceId) | GlobalFilter is OK |
| Authentication / Token validation | ✅ **Use Spring Security** |
| Authorization (roles) | ✅ **Use Spring Security** |
| Production-grade security | ✅ **Use Spring Security** |

**So yes: CustomFilter is not recommended for security in real-time projects — use Spring Security instead.**

**Coustom Filter Code Explanation :**

Sure! Let's explain your CustomFilter code **line by line** in a **very simple, easy way** — like you're explaining to a school student 👦👧.

### ✅ FULL SIMPLE EXPLANATION:

**@Configuration**

**public class CustomFilter implements GlobalFilter {**

This says: "Hey Spring! I'm creating a **custom filter** for Gateway. Use this class when any request comes to the gateway."

**Logger logger = LoggerFactory.getLogger(CustomFilter.class);**

This line creates a **log tool**. You can print messages to the console like:  
 “Hey, someone made a request!”

**@Override**

**public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {**

This is the **main method**. It runs for **every request** coming through the Gateway.  
It handles the request (exchange) and passes it forward (chain).

**ServerHttpRequest request = exchange.getRequest();**

This line gets the actual **request**.  
Think of it like: "What URL is the user calling? What headers did they send?"

**if (request.getURI().toString().contains("/api/student")) {**

**}**

This checks:  
 "Does the URL have **/api/student** in it?"  
✅ If yes → you can do something extra for student URLs (like logging, checking, etc.)  
❌ If no → it just skips.

But inside this if block, you didn't write any code yet. So this check is useless right now.

**logger.info("Authorization = " + request.getHeaders().getFirst("Authorization"));**

This just **prints the** Authorization **header** (usually a JWT token).  
Even if the token is missing or wrong, this code just prints it — it does **NOT stop the request**.

**return chain.filter(exchange).then(Mono.fromRunnable(() -> {**

This line says:  
 "Let the request continue to the microservice."  
 After the microservice responds, do something extra (like logging the response).

**ServerHttpResponse response = exchange.getResponse();**

This gets the final response returned by the microservice.

**logger.info("Post Filter = " + response.getStatusCode());**

This logs the final status code (like 200 OK, 404 Not Found, etc.).

**}));**

Ends the post-processing logic.

### 🔍 So what is this filter ****actually doing?****

Just two things:

✅ Logs the Authorization header from the incoming request.

✅ Logs the final status code after the request is finished.

### ❌ It does ****not****:

Block bad headers

Validate tokens

Reject wrong URLs

Add or remove anything from the request

✅ In very simple words:

This is just a **security camera** 📷 — watching and printing info about requests, but **not stopping anything**.

**=================== post-filter and pre-filter right ================**

Whatever we do in Spring Security (authentication, token check, etc.) is kind of like Pre-filter logic. And if we want to do something after the response comes, that's called Post-filter logic Right?

Yes! You’ve **understood the concept correctly**. Let me explain it in your own words more clearly and confirm your understanding:

### ✅ What are ****Pre-filter**** and ****Post-filter**** in simple real-time terms?

They are **concepts** used in **Spring Cloud Gateway** to:

Do something **before** the request goes to the actual microservice → **Pre-filter**

Do something **after** the microservice sends a response back → **Post-filter**

### 🎯 Example with Authentication and Logging:

#### 🔐 ****Spring Security**** handles:

Checking token (Authentication)

Validating role access (Authorization)

Usually done **before** hitting the microservice → works like a **Pre-filter**

So yes, **Spring Security logic is usually implemented using Pre-filters**, but it’s built-in — not manually written like Gateway filters.

### Logging or Adding Extra Data:

If you want to **log response**, or **modify response headers** after your microservice returns data — that’s a **Post-filter**.

### 🔁 Summary:

| **Feature** | **Handled by** | **Filter Type** |
| --- | --- | --- |
| JWT Token check | Spring Security | Pre-filter |
| Role-based access | Spring Security | Pre-filter |
| Logging request headers | Gateway filter | Pre-filter |
| Logging response status | Gateway filter | Post-filter |
| Add custom response header | Gateway filter | Post-filter |

### ✅ So your statement is correct:

"**Whatever we do in Spring Security (authentication, token check, etc.) is kind of like Pre-filter logic. And if we want to do something after the response comes, that's called Post-filter logic.**"

Yes! That’s exactly what **Pre-filter and Post-filter mean** in real-time projects.

// **Note** : E CustomFilter ane class just Pre-filter and Post-filter Concepect chudatanili rasamu anthe.  
 But real time project lo Spring Security ni implimnet cheyali  
  
  
 1. Pre-filter: means nothing but before sending the request to microservices if we done any  
 validation or any kind of oparation we will call that Pre-filter.  
  
 2. Post-filter: means nothing but after getting the response from microservices if we done any  
 changes to the response or any kind of oparation we will call that Post-filter.

======== Circuit Breaker using Resilience4j in microservices ==========

The request failed continusly means the next request it will not check any thing and it will give fallback mentod error responces directly right if it is correct means then how it will know if it is working means

**YES! You've got it!**  
Once the Circuit Breaker sees continuous failures and the threshold is crossed, it **opens the circuit** — and **stops making real HTTP calls**. It instantly calls the fallbackMethod().

But your next question is brilliant:

🧠 “If it's not making real requests anymore, how does it know the service is back up again?”

This is where the half-open **state** comes in. Let’s explain the **full lifecycle** 👇

## 🔄 Circuit Breaker Lifecycle (Resilience4j)

### 1. ****CLOSED**** (Normal State):

Everything is fine

All requests go to the real service

If too many failures in the last N requests → go to **OPEN**

### 2. ****OPEN**** (Short-Circuited):

No request goes to the real service

All requests are blocked immediately

Fallback is returned

**Timer starts** (wait-duration-in-open-state)

✅ **This saves your resources and avoids flooding the broken service**

### 3. ****HALF-OPEN**** (Trial Mode):

After wait-duration-in-open-state (e.g. 30 seconds), breaker tries a few requests

Controlled by:  
permitted-number-of-calls-in-half-open-state=5

### 👉 What happens here:

Resilience4j **lets 5 requests pass through** to the real service

If **enough succeed** → circuit goes **CLOSED**

If they **fail** again → circuit goes **OPEN** again

### 🔄 This cycle keeps repeating, automatically.

✅ So even when circuit is open, **Resilience4j will auto-check** if the downstream service has recovered using half-open mode.

## 📊 Real-Life Example Timeline

| **Time** | **State** | **Request** | **Action** |
| --- | --- | --- | --- |
| T1 | Closed | 10 requests | 6 failed → threshold crossed |
| T2 | Open | Requests 11–1000 | All short-circuited, fallback returned |
| T3 | 30 sec later | Half-open | 5 real requests allowed |
| T4 | If 4 succeed | → Circuit Closed again |  |
| T5 | If 3+ fail | → Circuit Open again |  |

## 🔧 From Your Properties:

resilience4j.circuitbreaker.instances.addressMicroserviceCommonService.sliding-window-size=10

resilience4j.circuitbreaker.instances.addressMicroserviceCommonService.failure-rate-threshold=50

resilience4j.circuitbreaker.instances.addressMicroserviceCommonService.wait-duration-in-open-state=30000

resilience4j.circuitbreaker.instances.addressMicroserviceCommonService.automatic-transition-from-open-to-half-open-enabled=true

resilience4j.circuitbreaker.instances.addressMicroserviceCommonService.permitted-number-of-calls-in-half-open-state=5

🔍 So in your case:

If **5 out of 10** fail → OPEN

Wait 30 sec → HALF-OPEN

Allow 5 real calls

If success rate is high → CLOSED again

## 🧠 Final Summary

| **Concept** | **What It Means** |
| --- | --- |
| Open Circuit | Service is unhealthy → block all requests, call fallback only |
| Half-Open | Test the service with limited real calls |
| Closed | Everything is fine, full traffic allowed |
| Auto-Recovery | Circuit Breaker tests downstream automatically |
| Benefit | Avoids flooding broken services, saves threads, improves resilience |

that means in Open state only it will not take any requiest and it will wait some time right remain stages it will take requirest and it will check -- only use for server down time no requiest will not accept right

✅ **Exactly! You're getting it 100% correct now!**  
Let me break it down very clearly and confirm your understanding point by point:

## 🔁 Circuit Breaker States — Behavior Summary

| **State** | **Are real requests sent to downstream service?** | **What does it return?** | **What's the purpose?** |
| --- | --- | --- | --- |
| **Closed** | ✅ Yes | Real response or error | Normal mode, tracking failure rate |
| **Open** | ❌ No | ❌ Instantly returns fallback | To **protect your service** from calling broken one |
| **Half-Open** | ⚠️ Only a few (e.g., 5) | Real response or fallback | To **test if downstream is back up** |

### 🔴 OPEN STATE — Your Focus Point

✅ Yes, in **Open State**, Circuit Breaker **completely blocks all real HTTP calls** to the downstream service.

It does **not try anything**

It just **returns fallback response**

It does this for a period like 30 seconds (wait-duration-in-open-state)

✅ This avoids:

Thread blocking

Timeouts

CPU/memory pressure

DDoSing the already broken service

## 📌 Your Understanding in Your Words:

"**Only in Open state it will not take any request and wait for some time, right?**  
**In other stages, it will take requests and check?**  
**So it is useful only during server down time to avoid requests going to the broken server, right?**"

### ✅✅✅ ALL 100% CORRECT ✅✅✅

## 🧠 Why It's Important Even Though You Return Error Anyway

### ❌ Without Circuit Breaker

Every call waits for **timeout** (e.g. 5 seconds)

All threads are stuck

Service calling it (like OrderService) also becomes slow

High chance of **cascading failure** (your entire system collapses)

### ✅ With Circuit Breaker

After threshold, returns fallback in **<10ms**

No HTTP timeout wait

Keeps your upstream service fast and responsive

Prevents **outage propagation**

## 🔄 Bonus: What Happens After 30s?

Circuit Breaker goes **Half-Open**

Sends **limited number of test requests** (like 5)

If success rate > threshold → **Closed**

If failures → back to **Open**

### So yes:

**Only "Open" blocks requests**

Others still make requests

The whole goal is to **avoid wasting time/resources when the downstream is already down**