**Microservices:**

To understand microservices, lets take the example of amazon app.

In amazon app, there are lot of diff services like payment, shipping, search etc..

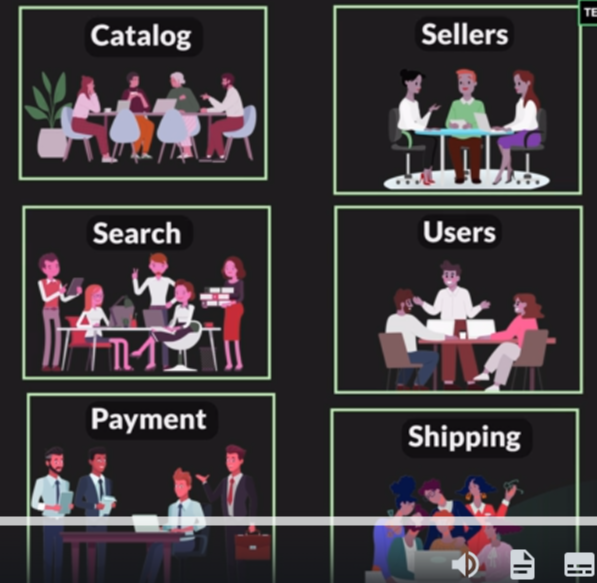


We will separate all the services into individual small parts like,

A screenshot of a computer

Description automatically generated

Then, lets say, in a project there are 50 members. We will separate them into 10 teams of each 5 members. Each team will work in a service individually.



But still this app will be a single package,



And we will create a war or jar file and we will put all the services in this war file and deploy that war file in a server or cloud.

Advantages: Entire app is in a single package. We call this as a monolithic application or monolithic architecture. Also we can deploy this as a single package.

Drawbacks:

(1)Team Dependencies: Every team working on individual services should have to be dependent on other services. For eg, when u have to release(update/deploy new feature), u have to be dependent on other services also to complete their update. Only then we can deploy it, as we can only deploy the entire project.

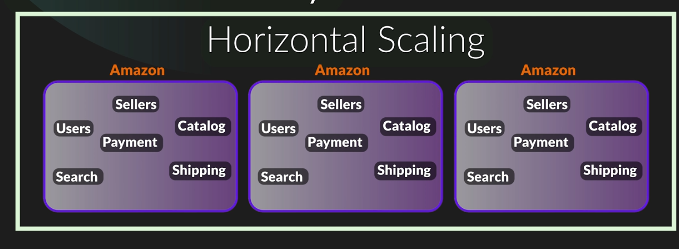
(2) Scalability: If a mega sale is coming on next week, then some services needs to be scaled so that it can manage the load. Eg, payment & searches will be high on a sale day. Then we need to scale those services. But since this app is in a single package, we can’t scale it individually. We can only scale the entire project.

How scaling is done:

1)Vertical Scaling:

 Here, we will add more servers or more RAM power.

2)Horizontal Scaling:



Here we will have multiple instances of the same app.

But in monolithic architecture, we can’t scale specific services.

(3)Another scalabitily drawback is **same technology**:

We can use only the same tech(java) on all services. We can’t use different languages for diff services. This is bcoz some language will be more efficient for some services.

**Microservices**:

Microservices is kind of architecture, where project is divided into diff services and all the services are self contained.

A screenshot of a computer

Description automatically generated

Each service can be deployed, scaled and can be developed on diff technology(language) separately.

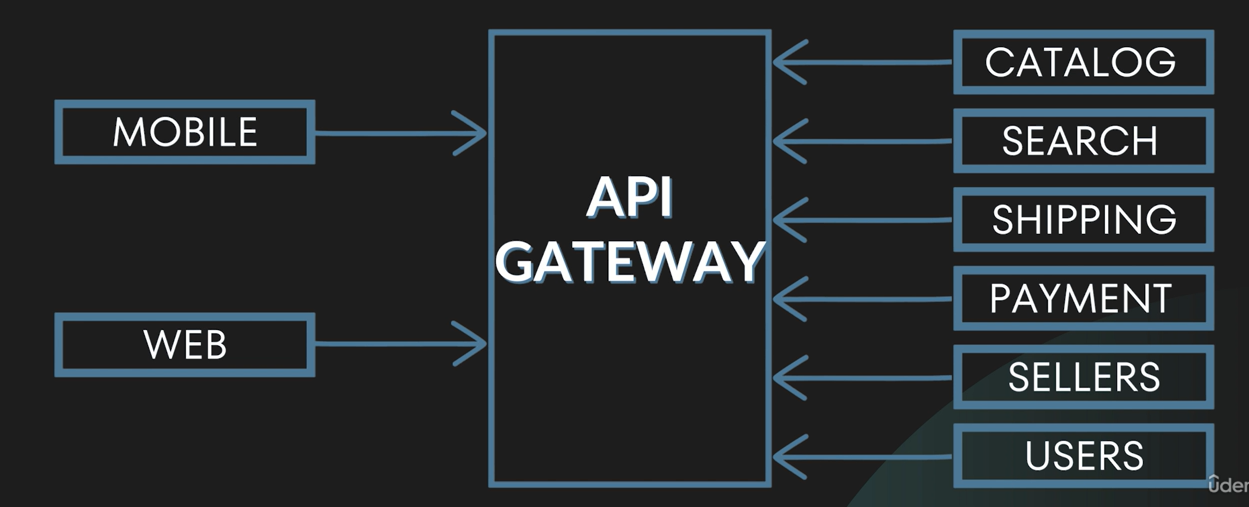


Each service here is called microservice, hence the name Microservices.

In Microservices, even if a single service failed it wont make the entire app down.

Two major things we have to work on microservices are communication btwn services and their security

Communication is managed by a API gateway and they communicate through HTTP request and response.



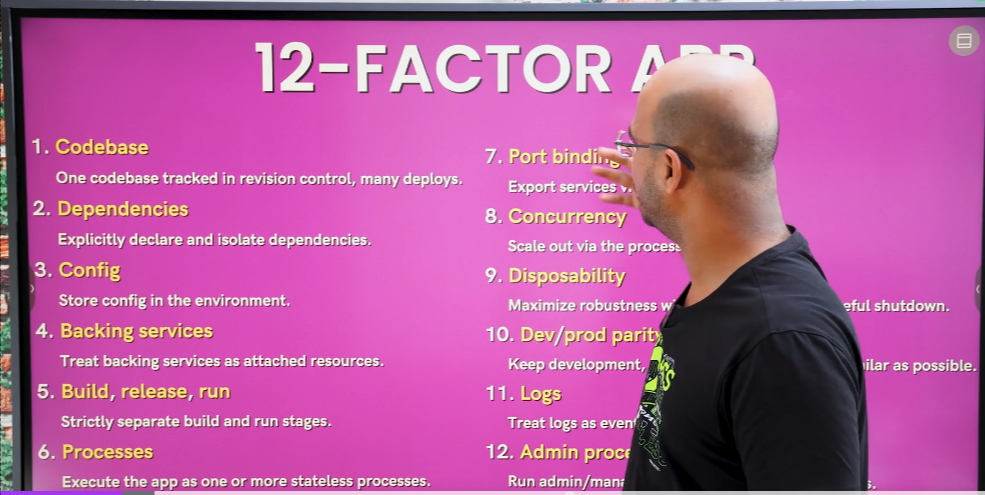
**Cloud Ready vs Cloud Native:**

**Why cloud**: Most of the companies now are deploying their applications in the cloud. Even though all companies have their own servers, most companies use cloud.

This is bcoz of the benefits of the cloud like cost, scalability and less issues.

Cloud Ready: Applications which are already completely developed and running on server. We will have to just make some changes and make it cloud ready.

Cloud Native: Applications(mostly new projects) when started to develop itself, will be developed as cloud native, which is nothing but developing the app using certain rules & regulations. This app is also called 12 factor app as it is developed using 12 rules.



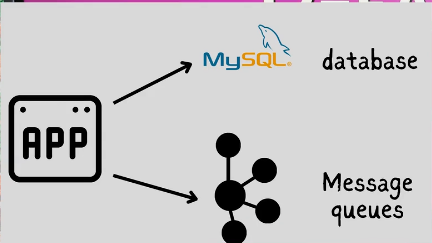
1.Codebase: Having one codebase for one application. Not one codebase for multiple apps or multiple codebases for one app.

2.Dependencies: All the maven dependencies will be stored in a separate folder called libraries. U don’t add the dependencies with base code itself. Also in the dependencies name and version should be mentioned, so that version issue can be avoided when you share it with testing and devops team.

3.Config: Configurations like for DB, you give url,username,password, port number etc..

All these configurations shld be given in separate file called .env file. So that, even if u change the server or something like that, you don’t need to change the codebase. So the main thing is we should not change the codebase.

4.Backing services: It’s nothing but loose coupling with any third party services like DB, Message queues etc.. Loose coupling means all these connections should be made in a separate file. For eg: If you are using MySQL now and if you have to change to PostGreSQL then it should be easy. We should not mention anything abt which DB resource in codebase.



5. Build,release and run: You can’t just create a app and run it. 3 steps are there for this. When you made a project ready, you have to build it into a package(maybe a jar or war file). You will copy the package and release it to the running environment, and this release will have a particular version to it. So even if the new added feature failed in this release, you can again go the older release version. So the advantage is, if we want to do a small change, we don’t need to do it in release version. We can simply do it in build version and release it.

6. Processes: Keep ur app stateless (i.e)which means when a client reuqests a data from server/db, it fetches the data and it remembers the user. So the next time, when the user requests a new data, the server remembers him and send him the data without verifying again who he is. This is called Sticky Session. But we should not do this. Whenever a data request comes from the client, it should just take the data from the DB. Advantage is, at any given point, if u stop the process, data will not be lost as it is stored permanently stored in DB.

7. Port binding: Every service should have a port number. We will not be sure which physical server the service is, but we know what type of service it provides. Helps us to find and identify the service.

8. Concurrency: If you want to increase scalability, then go for horizontal scaling(scale out). Vertical scaling(scale up) has a limit. You can’t just increase the RAM or processor. It has a certain limit. Instead you can create multiple instances of the same service.

A close-up of a computer scale

Description automatically generated

9. Disposability: Fast start- If ur app runs on DB, then when you start ur app, DB should be connected automatically. Graceful shutdown- When u close the app, all the connections(eg: DB connection) should be close properly. Also, even when a service failed or the app went down, the data should not be lost.

10. Dev/prod parity: Using containerization, we have to make the development, staging and production much faster and easier. For eg, Using docker, you can transfer the project to any OS, as every OS has docker software, and we can run the project on it easily.

11. Logs: Logs are very important to monitor our application. So we have to log everything properly to analyse and fix error if any issue arises in app.

12. Admin processes: We should be able to admin the app either using using some port or some services.

That’s it. These are the 12 steps to create a Cloud Native app. It is always better to start developing a app as Cloud Native App to get use of all the cloud benefits. Bcoz, it will be difficult if you already create a app and then change it to Cloud Native.

**Quiz App:**

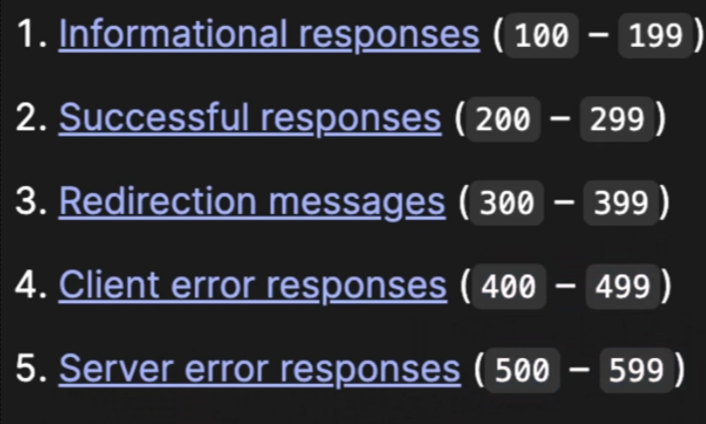
**Exception Handling:**

Till now we have not implemented exception handling. Now we have to handle the exception for all requests.

So, whenever a error rises we are going to handle it and send the error message(Status codes) to the client.

HTTP Response Status Codes:

These are the 5 Response Status Codes:



1. Informational responses – sends data
2. Successful responses- sends successful response
3. Redirection responses- redirects page
4. Client error responses- when client makes a mistake like entering wrong url
5. Server error responses- like, when there is a bug in code.

[200 OK](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200), [201 Created](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201), [400 Bad Request](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400), [401 Unauthorized](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401) are few of the HTTP error response status codes.

We don’t have to remember the number. We can just send the status codes alone to the client. We can achieve this by using ResponseEntity<> class, where we can return to the client both the data and the HTTP status codes.

Refer this url for more info, [HTTP response status codes - HTTP | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)

**Implementing Microservices in Quiz App:**

In Microservices, we will be having the services as separate individual services. (ie) quiz and question service will be individual services having its own DB).

So, we can’t access the question DB from quiz service through questionDao, like we did in monolithic app(initial quiz app. Refer Documents->Telusko Spring framework notes->QuizApp).

So, everything will be independent services. So, that we can scale a single service or update a single service and release them alone for production release without any depending on other services.

**Load Balancing:** Lets assume the client will be training 2000 students tomorrow and we have to scale up the quiz service and we create 10 instance of quiz service and 2 instance of question service. Here when a quiz service is requesting a question service, it will get confused which instance of question service to reach to. In this case, we need load balancing to sort it out.

**API Gateway:**

But since all the services are independent, each server have different IP address. So, URL’s will be different for all services. User don’t have to remember which service have which IP address. We just have to send the request to API Gateway and it will take care of it.

Here, API Gateway helps us to reach the desired services by acting as a lift lobby.

A purple rectangle with white arrows

Description automatically generated

**Service Registry:** Now lets say, we have 10 services. In this case, if 1st service has to access 6th service, it don’t know where the 6th service is. Here, Service Registry will play the role and helps the services.

**Microservices using Quiz App:**

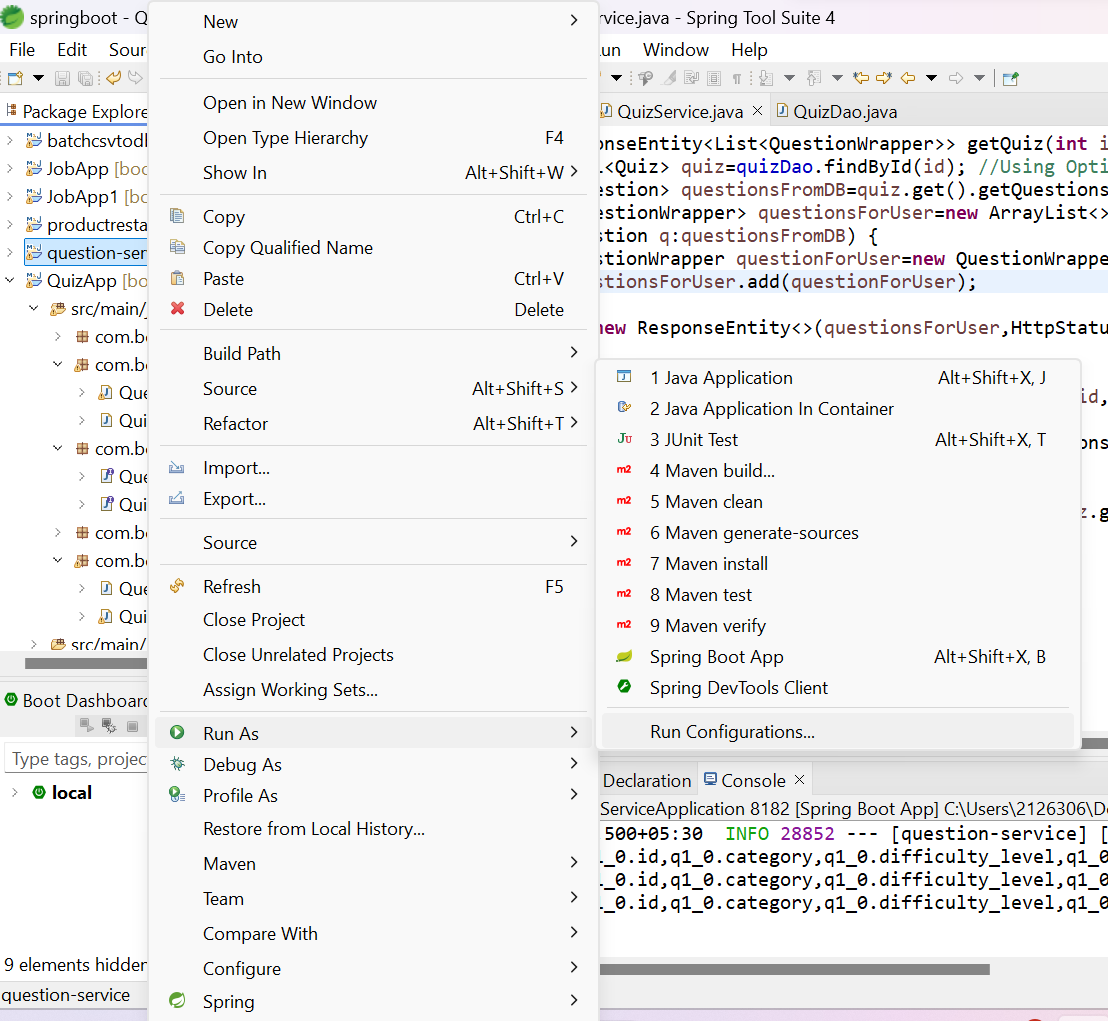
In our microservices of QuizApp, user will be only interacting with the quiz-service. The quiz-service only intercate with the question-service and fetch the required data needed by the user. This is bcoz, question-service only has the Question DB and quiz-service has no access to Question DB. So for both creating quiz(both generating question id’s and getting the questions of those generated id’s) and getting score, quiz-service have to request it to question service.

A person standing in front of a quiz

Description automatically generated

**How to create multiple instances of one service:**

Right click the service on left side in package explorer-> Run -> Run configurations



In Run Configurations, right click the service and select Duplicate.

A screenshot of a computer

Description automatically generated

After that, click the duplicate and go to Arguments, and in VM arguments, give a different port number.

A screenshot of a computer

Description automatically generated

Every instance should run on different port number.

How to call one server from another server:

We need two things to do this:

1. Service Discovery(Netflix Eureka/Eureka Server)
2. Open Feign

**Eureka Server:**

Every server should register itself in Eureka Server(Service Registry) and also it should have Eureka Client(name of that server).

**A diagram of service

Description automatically generated**

Open a new project named service-registry.

1.Add dependencies spring-web and eureka-server.

2.Add annotation @EnableEurekaServer in the proj main class.(this will make this eureka server).

In application.server, add these 5 properties.

A screenshot of a computer

Description automatically generated

Name and port is must to know where eureka server is present.

5th and 6th property,

A diagram of service

Description automatically generated

Usually, eureka clients will be registered in eureka server and by registering the client we can access them. Likewise, we should not register the eureka server itself and also shld not fetch the server.

SO, 5th property represents that we shld not fetch the server and 6th property represents that we shld not register the eureka server.

Also, the 1st property is important as it will make the eureka server work.

In 2nd property, the port number 8761 is the default port for eureka server.

That’s it. Eureka server is ready.

**Eureka Client:**

Now, to make eureka client register in eureka server, we have to do two things,

1. In application.properties, add application name for the microservice.(spring.application.name=question-service)
2. In pom.xml, add the dependency, spring-cloud-starter-netflix-eureka-client.

That’s it. The microservice will get registered in the eureka server.

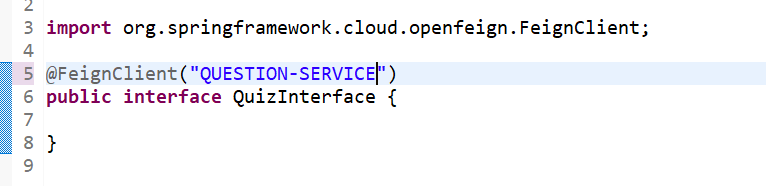
**Open Feign:**

Open Feign allows us to access another server just by mentioning the server name(Eureka Server). We don’t need IP address(localhost) or port number(8181).

There is a class called RestTemplate to call other microservices but we need to mention ip address and port number. It is not a good practice to use ip address and port number explicitly. That will tight couple the microservices.

Once eureka server is set up, we need to setup Feign Client.

1. Add open feign dependency in pom.xml
2. Need to add @EnableFeignClients in quiz-service main class as We
3. Create a interface for the service(any name) and add the annotation @FeignClient(“service-name”) and inside the brackets, give the service name that you want to access.



The service name should be same as you gave in the property file. Bcoz that name will be taken in eureka server in capital letters.

A screenshot of a computer

Description automatically generated

1. Now add the methods of the question-service, that you need to access.

From quiz-service, we only need 3 methods from question-service. Generate() to generate the question id’s, getQuestions() to get the questions and getScore() to generate the questions.

A screenshot of a computer screen

Description automatically generated

So we need to declare these 3 methods alone in the QuizInterface. No need to define the methods as they are already defined in question-service.

Now, we need to call these methods in the QuizService class using the QuizInterface reference.

This is the way(using a interface and giving the annotation @FeignClient) through which we can access other services from a service.

Note: Eureka Client is for registering the service in service-registry so that the service can be discovered and accessed by other services and also this service can access other services.

@FeignClient helps in simplifying the communication between services by allowing to access other services without the hostname and port number and just by mentioning the target service name in @FeignClient(“target-servicename”).

**Load Balancing:**

Lets say, if question-service has 3 instances(horizontal scaling) and 1st and 3rd instances are busy. When quiz-service is requesting for the question service, it will access instance of question-service which is not busy(which has less load).

This process is called load balancing. And it is done by Load balancer.

To implement this in our microservices, we don’t need to configure anything separately. Eureka server and open feign dependency itself will add load balancers jars in libraries.

When you add @FeighClient(“service-name”) annotation and mention the microservice in brackets, it will automatically search for the instance which has less load and request to it.

**API Gateway:**

In microservices, we have many services and each services have their own port number. And only by using that port number, we can use that service and access other services.

But the user dont know all these microservices and their port numbers. According to user, it is a single application and he should access that app with a single URL.

So we have to make it to one single URL, which user can work.

We can achieve this with the help of API Gateway. API Gateway acts as the entry point to the app or it acts as the interface btwn the user and the app.

For this, we need to create a new project with dependencies Gateway from spring cloud router and eureka client bcoz we need to register api gateway in eureka server.

Adding eureka client dependency itself will make in register in eureka server.

Now, we have to make 4 changes in application properties.

A screenshot of a computer

Description automatically generated

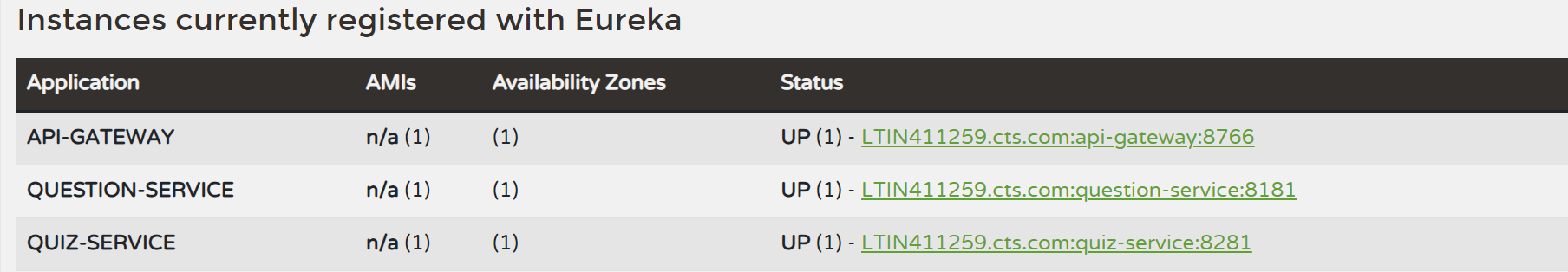
1. Server.port 8765 will now be the common url for all services.
2. spring.cloud.gateway.discovery.locator.enabled=true. This will make the api-gateway to search for the service by seeing the service-name that we give in the URL after port number(quiz-service). This is an important property for api-gateway to work.
3. spring.cloud.gateway.discovery.locator.lower-case-service-id=true. Defaultly service-name will be stored in capital in eureka server. If we need to give it in small letters in URL, then we need to add this property.
4. eureka.instance.prefer-ip-address=true. We need to give this property in the property file of all the services to avoid DNS error.

In postman,



Now, you can give the port number 8765 itself for all requests. But the api-gateway don’t know which service we are requesting.

The only way for the api-gateway to know which service we are requesting is through eureka server. Bcoz all the services are registered in eureka server.



So, we need to take the service/instance name from eureka server and give it in the URL after the port number like this,

 So that when we request the quiz-service URL with api-gateway port number also, it will identify that the request should go to quiz-service.

Note: We can also configure API gateway in a way that we don’t have to mention the service-name also in the URL. But that involves quite a lot of configurations and it comes under advanced concepts.

Microservices projects: Question-service,quiz-service,service-registry,api-gateway(total 4 projects).