We are calling PostService from CommentService and vice versa by hardcoding the URL through RestClient. This approach works fine on our local machine because services usually run on fixed ports like 8080. But in a **cloud environment**, services are deployed dynamically with different ports and IP addresses, which are not fixed or predictable.

In such environments, we can have **multiple instances** of PostService and CommentService. Suppose CommentService is deployed on 3 instances, each with different dynamic IPs. Now, if PostService wants to call CommentService, how will it know **which IP to use**? Even if we hardcode the IP of one instance, that instance might go down — so how would the communication happen with the remaining instances?

To solve this issue, we use the **Service Discovery Pattern**.

**What is Service Discovery Pattern?**

Service Discovery means creating a **central registry server** (like **Eureka**) where all microservices **register themselves** at startup. This central server stores the IP addresses and metadata of all service instances. We call this internal list a **service registry**.

So now, when PostService wants to call CommentService, it doesn't hardcode the address. Instead, it **queries the discovery server** (Eureka) and asks, *“Where is CommentService running?”* The discovery server replies with one or more instance addresses, and PostService uses one of those to make the actual request.

This way, we **avoid hardcoding** URLs in both PostService and CommentService.

**Client-Side Load Balancing**

When a service (like PostService) communicates with the discovery server for the first time, the **entire registry** is sent back. The service client then **stores this registry locally**. So, even if the discovery server (Eureka) becomes temporarily unavailable, the service can still make requests using its **cached registry**.

Now imagine CommentService has 3 instances. If PostService tries the first instance and it's down, it automatically tries the second. If that fails, it tries the third. If all three fail, then it throws an error. This process is known as **client-side load balancing**, where the client (like PostService) is responsible for selecting and calling a healthy instance.

Note: Tools like **Spring Cloud LoadBalancer** or previously **Ribbon** handle this behavior internally.

**Final Thought**

By using service discovery and client-side load balancing, we make our microservices **resilient**, **scalable**, and **dynamic** — without depending on hardcoded IPs or ports. This is essential for running microservices in modern cloud environments.