NEED OF EUREK SERVER IN MICROSERVICES

Scenario: Online Retail Application

Imagine you have an online retail application with several microservices:

- **Product Service**: Manages product details.
- **Order Service**: Handles orders.
- **Inventory Service**: Manages stock levels.
- **Notification Service**: Sends notifications to customers.

Each of these services runs multiple instances for load balancing and fault tolerance.

Without Eureka Server

1. **Static Configuration**:
    - Each service needs to know the network address of the other services. For example, the Order Service needs to know where the Inventory Service is located.
    - If an instance of the Inventory Service is added or removed, the configuration must be manually updated and redeployed.
2. **Scalability Issues**:
    - Scaling up (adding more instances) or down (removing instances) is difficult because every change requires updating configurations across all services.
3. **Fault Tolerance**:
    - If an instance of the Inventory Service goes down, the Order Service may still try to communicate with it, leading to errors and possible downtime.

With Eureka Server

1. **Dynamic Service Discovery**:
    - All instances of each service register themselves with the Eureka server when they start up.
    - The Order Service does not need to know the specific addresses of the Inventory Service instances. It simply queries the Eureka server to get the list of available Inventory Service instances.

    Example:

    ```
    java
    Copy code
    @Autowired
    private DiscoveryClient discoveryClient;

    public List<ServiceInstance> getInventoryServiceInstances() {
        return discoveryClient.getInstances("inventory-service");
    }
    ```

2. **Automatic Load Balancing**:

- The Order Service can use a client-side load balancer like Ribbon, which retrieves the list of Inventory Service instances from Eureka and distributes requests among them.

Example:

```
java
Copy code
@Autowired
private LoadBalancerClient loadBalancer;

public void placeOrder(Order order) {
    ServiceInstance instance = loadBalancer.choose("inventory-service");
    URI inventoryServiceUri = instance.getUri();
    // Use the URI to communicate with the Inventory Service
}
```

3. **Scalability**:
    - When new instances of the Inventory Service are started, they register themselves with Eureka automatically.
    - The Order Service will discover the new instances on its next query to Eureka, enabling seamless scaling.
4. **Fault Tolerance**:
    - If an Inventory Service instance goes down, it stops sending heartbeat signals to the Eureka server.
    - The Eureka server removes the downed instance from its registry, and the Order Service no longer tries to communicate with it.

Example:

```
java
Copy code
@Scheduled(fixedRate = 5000)
public void checkInventoryServiceHealth() {
    List<ServiceInstance> instances =
discoveryClient.getInstances("inventory-service");
    if (instances.isEmpty()) {
        // Handle the scenario where no instances are available
    }
}
```

Real-Time Example Workflow

1. **Service Registration**:
    - The Inventory Service starts and registers with the Eureka server:

      ```
      yaml
      Copy code
      eureka:
        client:
          service-url:
            defaultZone: http://eureka-server:8761/eureka/
      ```

2. **Service Discovery**:

- The Order Service queries Eureka to find Inventory Service instances:

```
java
Copy code
List<ServiceInstance> instances =
discoveryClient.getInstances("inventory-service");
```

3. **Load Balancing**:

    - The Order Service uses Ribbon to load balance requests:

```
java
Copy code
ServiceInstance instance = loadBalancer.choose("inventory-service");
```

4. **Fault Tolerance**:

    - If an Inventory Service instance fails, the Eureka server removes it from the registry:

```
java
Copy code
@Scheduled(fixedRate = 5000)
public void checkInventoryServiceHealth() {
    List<ServiceInstance> instances =
discoveryClient.getInstances("inventory-service");
    if (instances.isEmpty()) {
        // Handle the scenario where no instances are available
    }
}
```

This example demonstrates how an Eureka server simplifies service discovery, load balancing, scalability, and fault tolerance in a microservices architecture, ensuring a more resilient and manageable system.