Need of Service Discovery

After removing the question-related code from our Quiz Service, we need to make some modifications to ensure it works effectively as a standalone microservice. These changes will help us establish proper communication between our services.

👉 Creating a Data Transfer Object

First, we'll simplify how data is passed to our API by creating a Data Transfer Object (DTO):

```
package com.telusko.quizservice.model;

import lombok.Data;

@Data
public class QuizDto {
    String categoryName;
    Integer numQuestions;
    String title;
}
```

This **QuizDto** class encapsulates all the information needed to create a quiz in a single object, making our API more organized and easier to use. Instead of passing multiple parameters separately, clients can now send a single JSON object.

👉 The Communication Challenge

In our monolithic QuizApp, creating a quiz was straightforward, the QuizService directly accessed the QuestionDao to fetch questions from the database. However, in our microservices architecture, this approach no longer works because:

- The Quiz Service doesn't have direct access to the question database
- The Question Service manages all question-related operations
- Services need to communicate over the network

👉 The Initial Approach: RestTemplate

One way to solve this would be using Spring's RestTemplate to make HTTP requests from the Quiz Service to the Question Service:

```
// Example approach using RestTemplate
RestTemplate restTemplate = new RestTemplate();
String url = "http://localhost:8080/question/generate";
// Make HTTP request to Question Service
```

👉 The Problems with Direct URLs

However, this approach has several significant limitations:

- **Hardcoded URLs**: We're specifying **localhost:8080** which assumes both services are on the same machine
- **Static Port Numbers**: If the Question Service changes its port, our code breaks
- **No Flexibility**: What if we have multiple instances of the Question Service running on different machines?
- **No Resilience**: If a service moves or changes, our application fails

These limitations go against the flexibility and resilience that microservices should provide.

👉 The Solution: Service Discovery

To overcome these challenges, we need two key components:

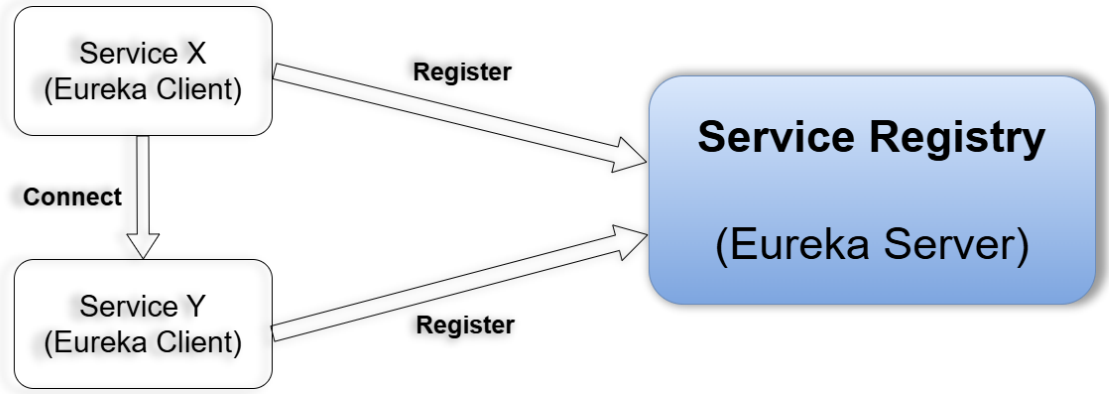
- **Feign Client**: A declarative way to call other services without hardcoding URLs and handling HTTP details
- **Service Discovery**: A mechanism for services to find each other dynamically

👉 Eureka Server for Service Discovery

Netflix's Eureka Server provides an elegant solution to our service discovery needs:

- ➢ We set up a central Eureka Server
- ➢ Each microservice registers itself with the Eureka Server on startup

- When one service needs to communicate with another, it asks the Eureka Server for the location
- Services can be found by their names rather than specific URLs



👉 **Benefits of Service Discovery**

- **No Hardcoded URLs**: Services refer to each other by name (e.g., "question-service")
- **Dynamic Discovery**: The actual location (IP and port) is resolved at runtime
- **Multiple Instances**: We can have multiple instances of each service and they're all discoverable
- **Automatic Updates**: If a service moves or restarts with a new IP or port, it re-registers itself
- **Load Balancing**: Requests can be distributed across multiple instances of the same service

By implementing Feign Client with Eureka Server, our Quiz Service can communicate with the Question Service without knowing its specific location. This makes our microservices architecture more robust, flexible, and truly distributed.