Load Balancing

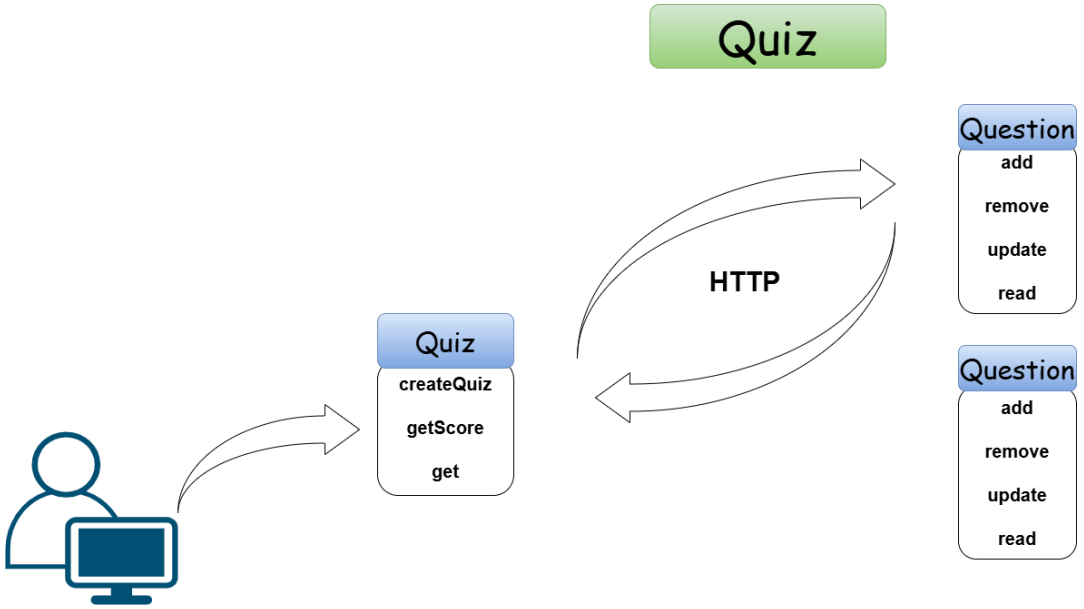
👉 Load Balancing

In modern applications, handling high traffic is a critical requirement. Load balancing is a technique that distributes incoming requests across multiple instances of a service to ensure:

- No single instance becomes overwhelmed
- Resources are used efficiently
- High availability is maintained
- Application performance remains consistent

From Monolith to Microservices Load Balancing

In monolithic applications, load balancing involves creating multiple instances of the entire application (horizontal scaling) and distributing requests among them. With microservices, we apply this same principle but with greater flexibility, we can scale individual services independently based on their specific load requirements.



Why Scale Individual Services?

Different services in a microservices architecture may experience different levels of demand:

- Some services might be frequently accessed (like a Question Service)
- Others might have less traffic (like an administrative service)
- Certain services might be computationally intensive
- Some services might handle time-sensitive operations

👉 Load Balancing in Spring Cloud

Modern Spring Cloud applications come with built-in load balancing capabilities:

- When using Eureka for service discovery
- When using OpenFeign for service communication

The combination of these technologies automatically enables client-side load balancing without requiring additional configuration.

👉 How Client-Side Load Balancing Works

When one service (like our Quiz Service) needs to call another service with multiple instances (like our Question Service):

1. The calling service queries Eureka to find all available instances
2. The load balancer component selects one instance based on its strategy
3. The request is sent to the selected instance
4. On subsequent requests, different instances may be selected

By default, Spring Cloud uses a Round Robin strategy, which distributes requests sequentially across all available instances.

Round Robin Strategy

- Round Robin is one of the simplest and most widely used load balancing strategies.
- Round Robin distributes requests sequentially across all available service instances, one after another in a circular order.
  - Imagine you have 3 servers: Server A, Server B, and Server C
  - The first request goes to Server A
  - The second request goes to Server B
  - The third request goes to Server C
  - The fourth request goes back to Server A
  - And the pattern continues in this circular fashion
- **In our microservices example**: When our Quiz Service needs to communicate with the Question Service (which has multiple instances), the requests alternate between the instances running on ports 8080 and 8081 in a predictable back-and-forth pattern.

Demonstrating Load Balancing

To verify that load balancing is working, we can modify our Question Service to report which instance is handling each request. This helps us visualize the load distribution.

```java
package com.telusko.questionservice.controller;

import com.telusko.questionservice.model.Question;
import com.telusko.questionservice.model.QuestionWrapper;
import com.telusko.questionservice.model.Response;
import com.telusko.questionservice.service.QuestionService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("question")
public class QuestionController {

    @Autowired
    QuestionService questionService;

    // Inject the Environment to access application properties
    @Autowired
    Environment environment;

    @GetMapping("allQuestions")
    public ResponseEntity<List<Question>> getAllQuestions(){
        return questionService.getAllQuestions();
    }

    @GetMapping("category/{category}")
    public ResponseEntity<List<Question>> getQuestionsByCategory(@PathVariable String category){
        return questionService.getQuestionsByCategory(category);
    }

    @PostMapping("add")
    public ResponseEntity<String> addQuestion(@RequestBody Question question){
        return questionService.addQuestion(question);
    }

    @GetMapping("generate")
    public ResponseEntity<List<Integer>> getQuestionsForQuiz
            (@RequestParam String categoryName, @RequestParam Integer numQuestions ){
        return questionService.getQuestionsForQuiz(categoryName, numQuestions);
    }

    @PostMapping("getQuestions")
    public ResponseEntity<List<QuestionWrapper>> getQuestionsFromId(@RequestBody List<Integer> questionIds){
        // Print the port number to identify which instance is handling the request
        // This helps us confirm that requests are being distributed
        System.out.println(environment.getProperty("local.server.port"));
        return questionService.getQuestionsFromId(questionIds);
    }

    @PostMapping("getScore")
    public ResponseEntity<Integer> getScore(@RequestBody List<Response> responses)
    {
        return questionService.getScore(responses);
    }
}
```

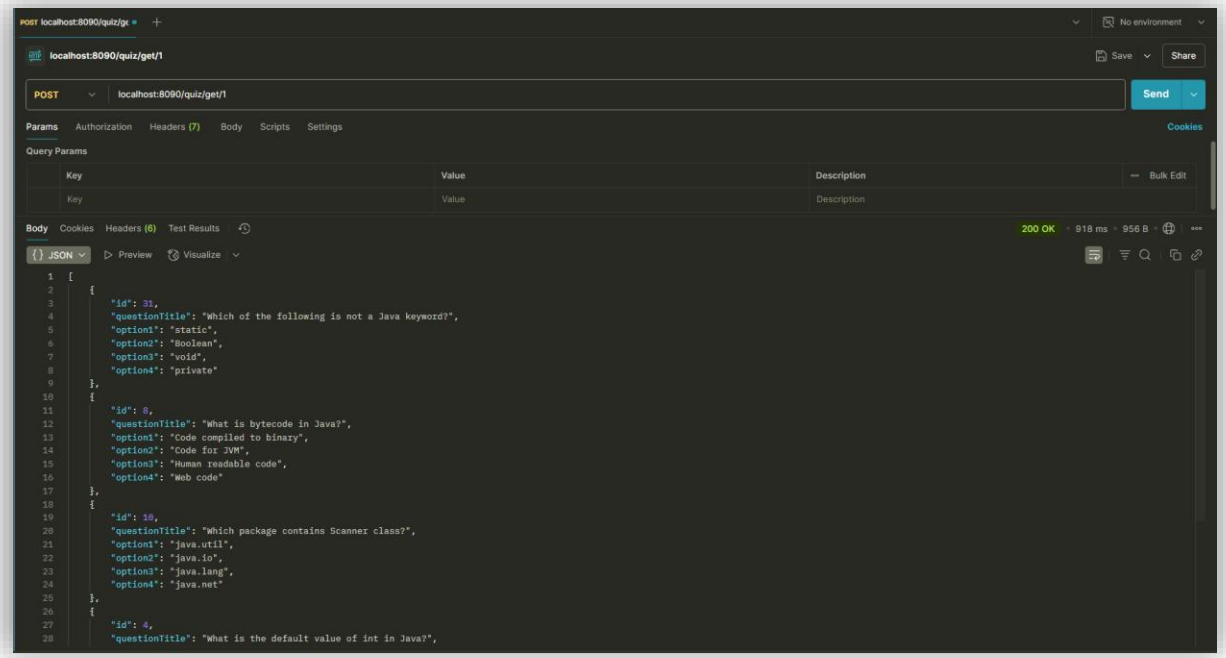
By adding the System.out.println(environment.getProperty("local.server.port")) line, we can observe which instance (identified by its port) is handling each request.

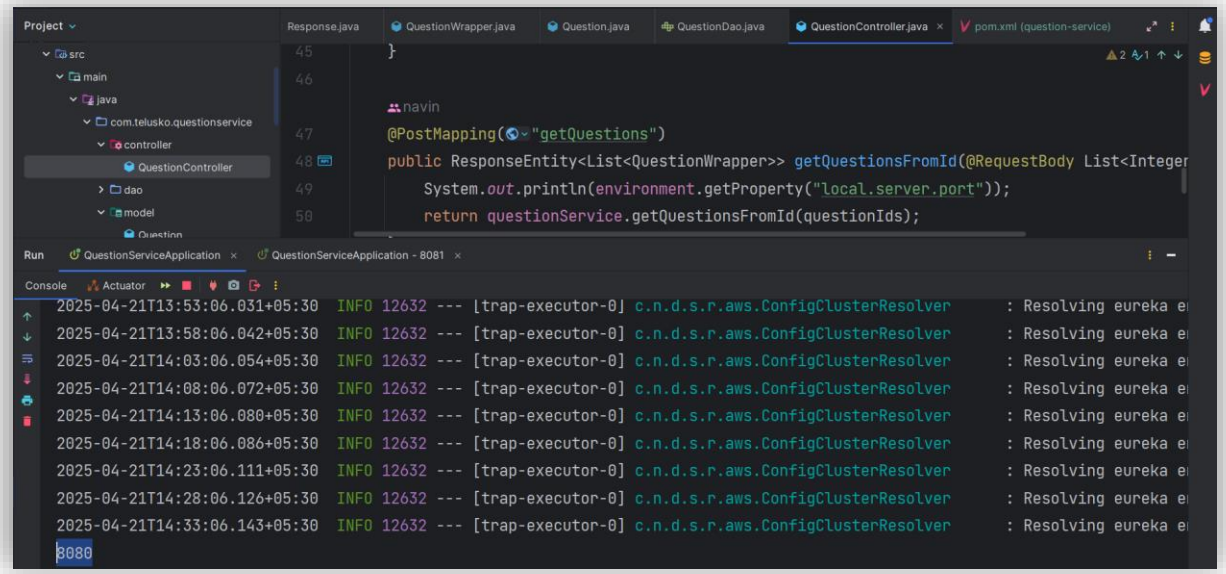
Testing Load Balancing

To test our load balancing, we need to:

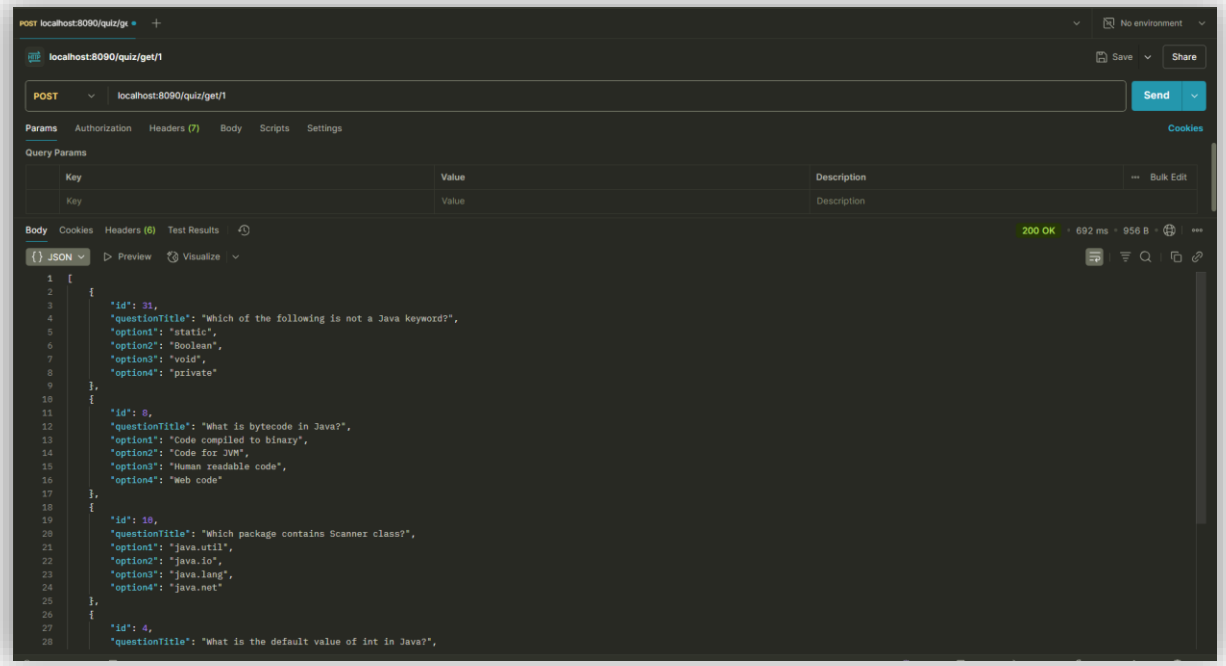
- Start multiple instances of the Question Service (on ports 8080 and 8081)
- Send multiple requests to the Quiz Service to retrieve questions
- Observe the console output to see which instance handles each request

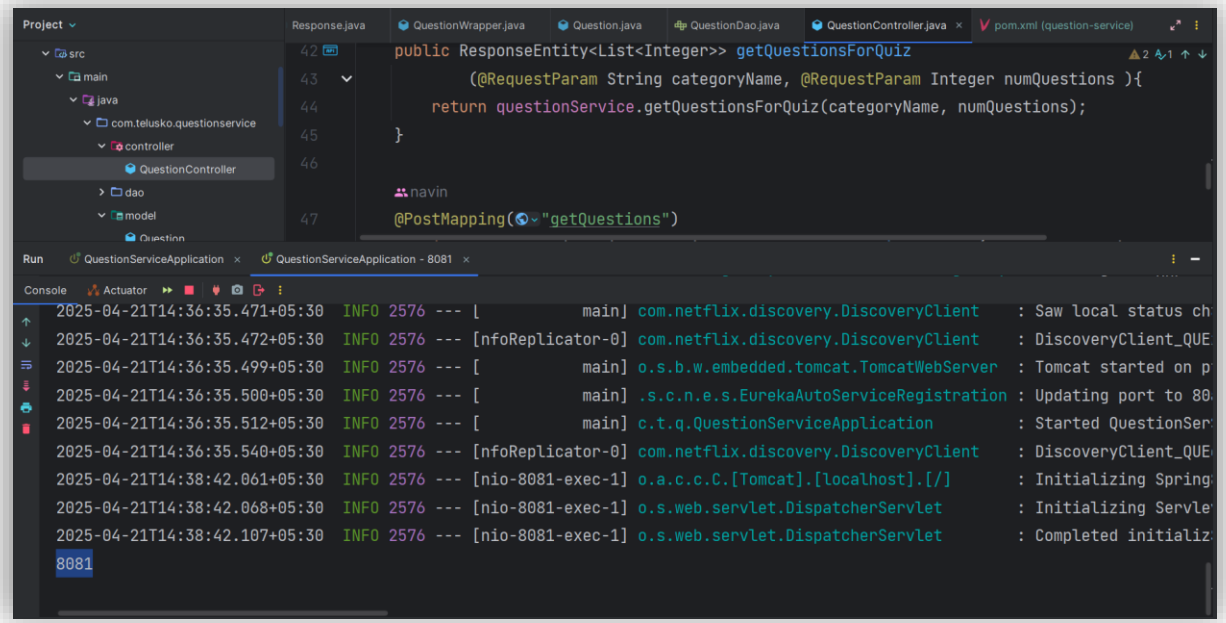
First Request





Second Request





By observing the console output, we can see that requests are alternating between our instances running on different ports. This confirms that client-side load balancing is working correctly, distributing requests in a round-robin fashion.

👉 Benefits of Load Balancing in Microservices

This automatic load balancing provides several key advantages:

- **Improved Scalability**: Handle more traffic by simply adding more instances
- **Better Resilience**: If one instance fails, traffic automatically routes to healthy instances
- **Resource Optimization**: Distribute load evenly across all available resources
- **No Single Point of Failure**: Multiple instances ensure continued operation even during partial outages
- **Zero Configuration**: Spring Cloud handles load balancing automatically