**Monolithic Architecture**

A monolithic architecture is a traditional software architecture model where the entire application is built as a single unit. In a monolithic application, all the components like the user interface, business logic, and data access layer are tightly coupled and run as a single process.

**Benefits:**

* Simplicity: Easier to develop and deploy initially.
* Performance: Can be more efficient since there's less inter-process communication.
* Unified Codebase: Easier to maintain in smaller applications.

**Drawbacks:**

* Scalability Issues: Difficult to scale individual components independently.
* Maintenance Challenges: As the application grows, it becomes harder to manage and update.
* Deployment Risks: A single change can affect the entire application, making deployments riskier.

**Microservices:**

Microservices is an architectural style where an application is structured as a collection of small, loosely coupled and independently deployable services. Each service is responsible for a specific business functionality and communicates with others using lightweight protocols like HTTP, REST or messaging queues.

**Purpose of Microservices**

* Scalability: Services can be scaled independently.
* Flexibility: Each service can use different technologies.
* Fault Tolerance: Failure of one service doesn’t affect the entire application.
* Faster Development: Small teams can develop, deploy and update services independently.
* Continuous Deployment: Easier integration with CI/CD pipelines.

**Microservices communicate with each other to exchange data and perform tasks.**

**There are two main types of communication:**

**1. Synchronous Communication (Direct Request-Response)**

  One service sends a request, and another service responds immediately.

Example: Service A asks Service B for customer details and Service B responds with the data.

**Common Methods:**

* REST API using RestTemplate (Manual HTTP calls)
* Feign Client (Simpler way to call other microservices)

**2. Asynchronous Communication (Event-Driven Messaging)**

Services don’t wait for an immediate response. Instead, they send a message and continue working.

   Example: When an order is placed, a message is sent to a queue, and another service (like Payment Service) picks it up when it's ready.

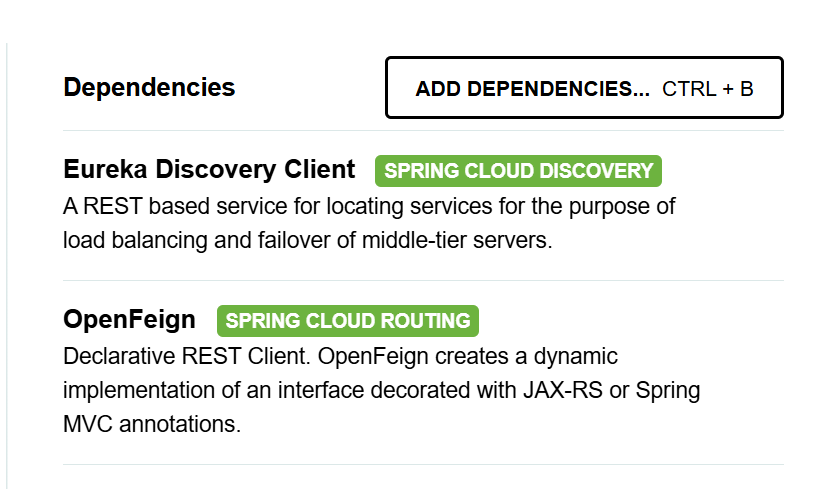
**Common Methods:**

     Kafka / RabbitMQ (Messaging Queue)

**Key Components of Microservices Architecture**

1. **API Gateway:** Acts as a single entry point for all clients and routes requests to appropriate services.
2. **Service Registry & Discovery:** Helps in locating services dynamically (e.g., Eureka).
3. **Configuration Management:** Centralized configuration management for microservices (e.g., Spring Cloud Config).
4. **Inter-Service Communication:** REST APIs or messaging queues (Kafka, RabbitMQ).
5. **Load Balancing:** Distributes traffic efficiently (e.g., Ribbon, Spring Cloud LoadBalancer).
6. **Security:** OAuth2, JWT, API keys.
7. **Observability:** Logging, Monitoring, and Tracing (ELK Stack, Prometheus, Zipkin).

**Add dependency:**



**Creating a service registry:**

**For Eureka Server:**

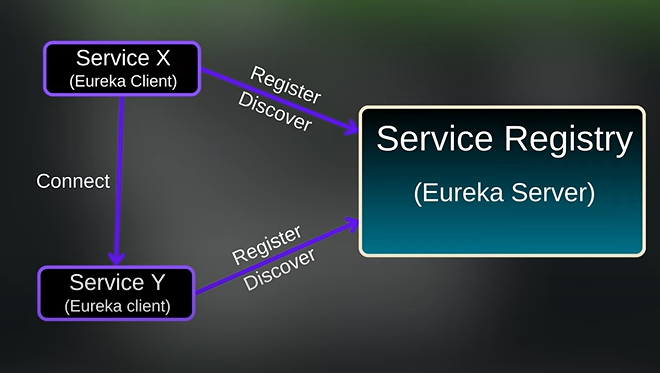
Eureka Server is an application that holds the information about all client-service applications. Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

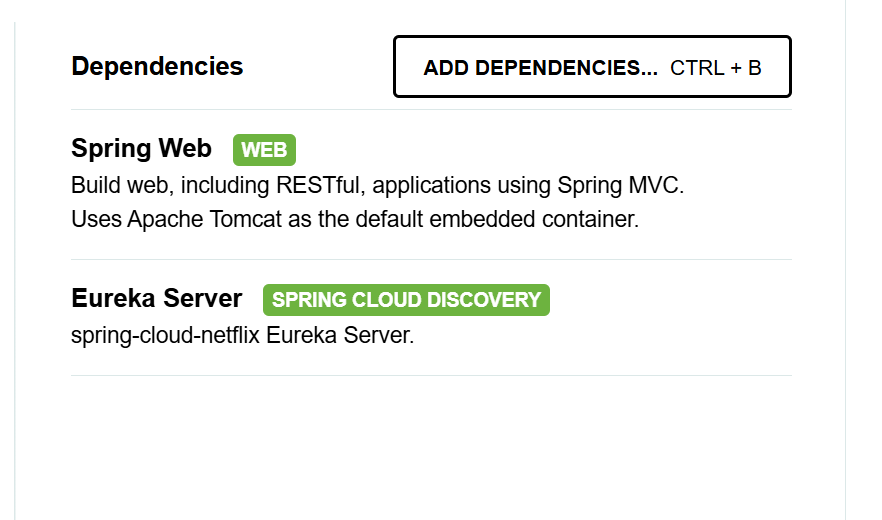
Think of Eureka Server as a phone directory for microservices. Instead of hardcoding service addresses, Eureka dynamically manages them.

**How Eureka Works:**

1. Each microservice registers itself with Eureka.

2. When one service needs another, it asks Eureka instead of using a fixed URL.





@SpringBootApplication

@EnableEurekaServer

public class ServiceRegistryApplication {

public static void main(String[] args) {

SpringApplication.run(ServiceRegistryApplication.class, args);

}

}

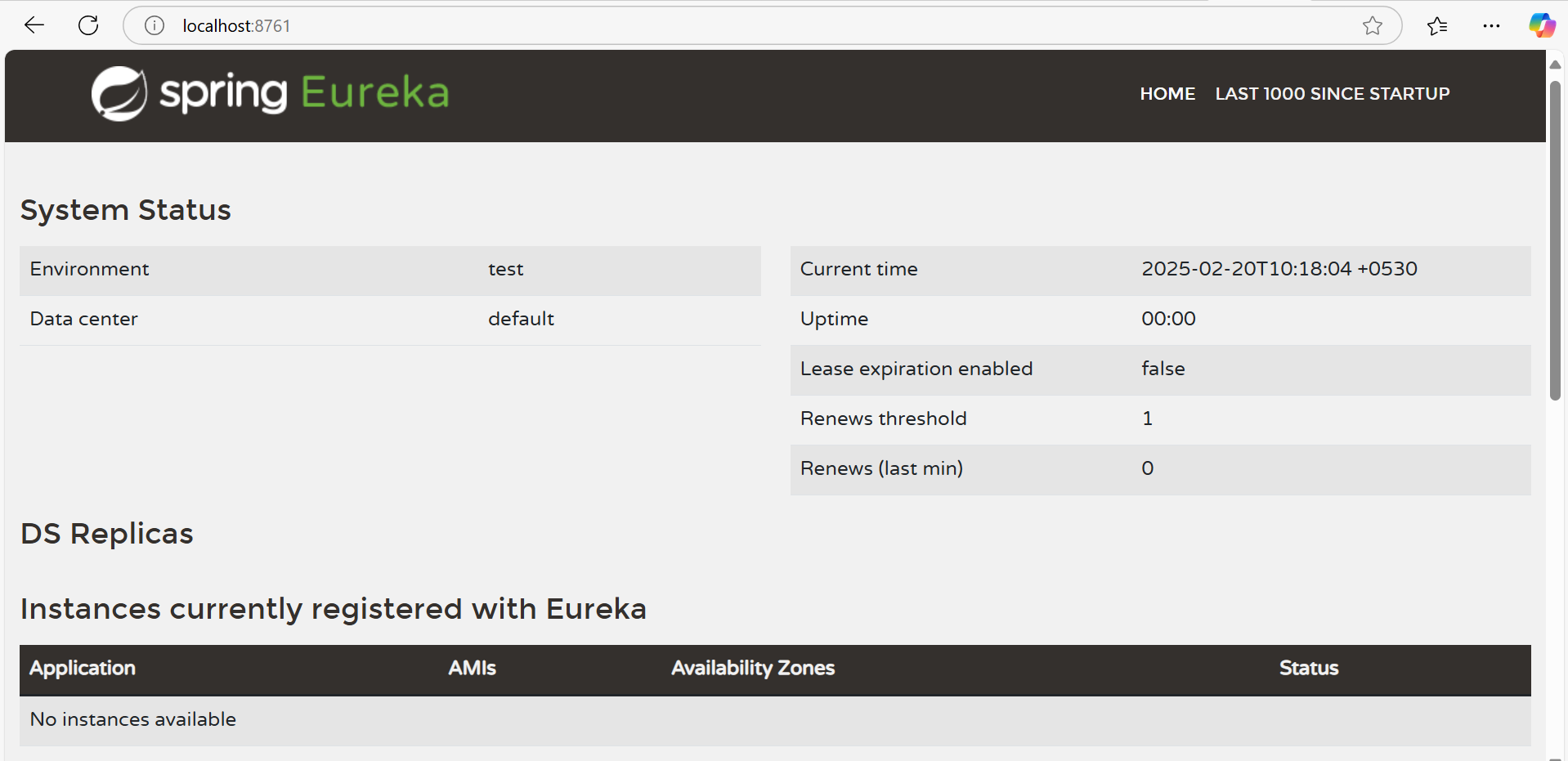
spring.application.name=service-registry

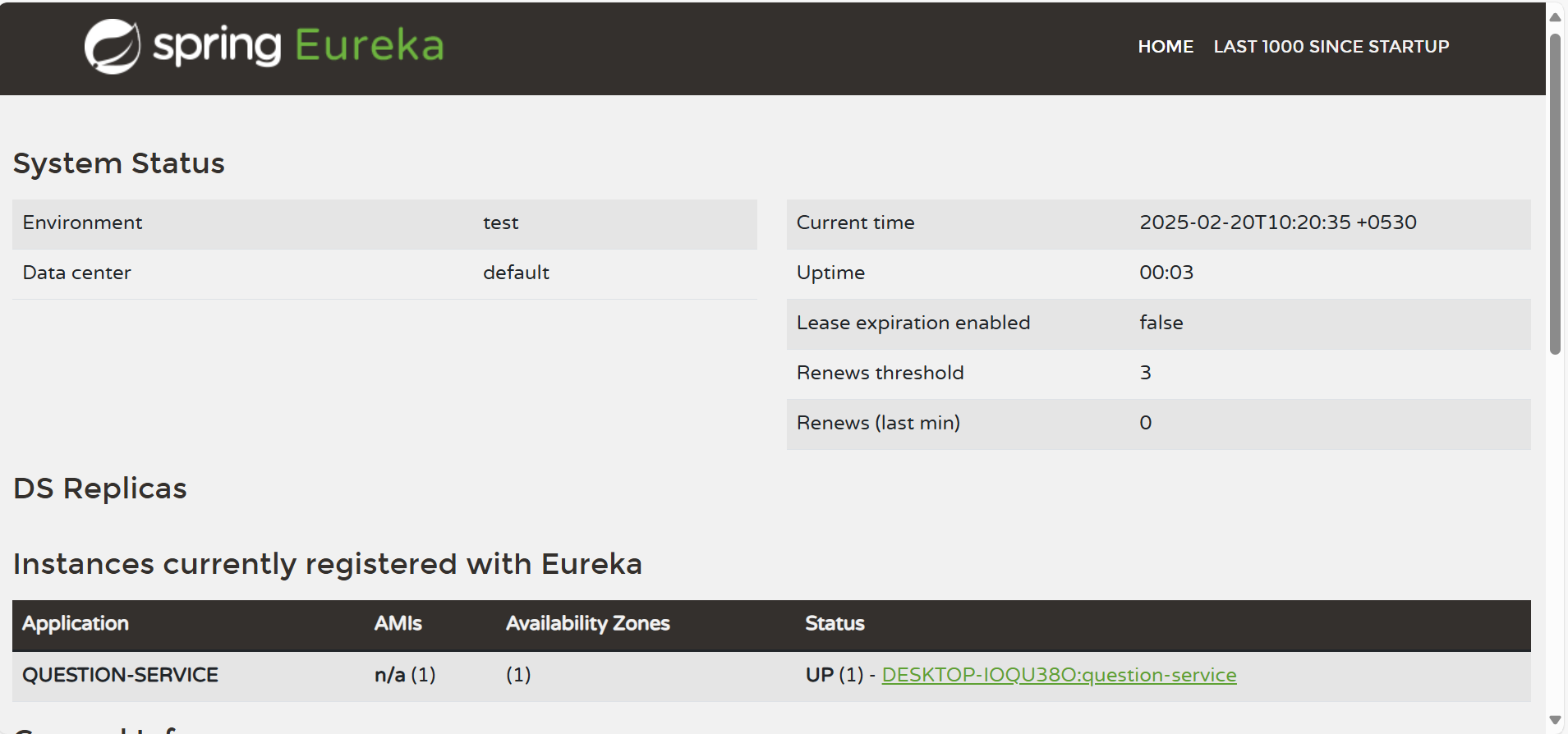
server.port=8761

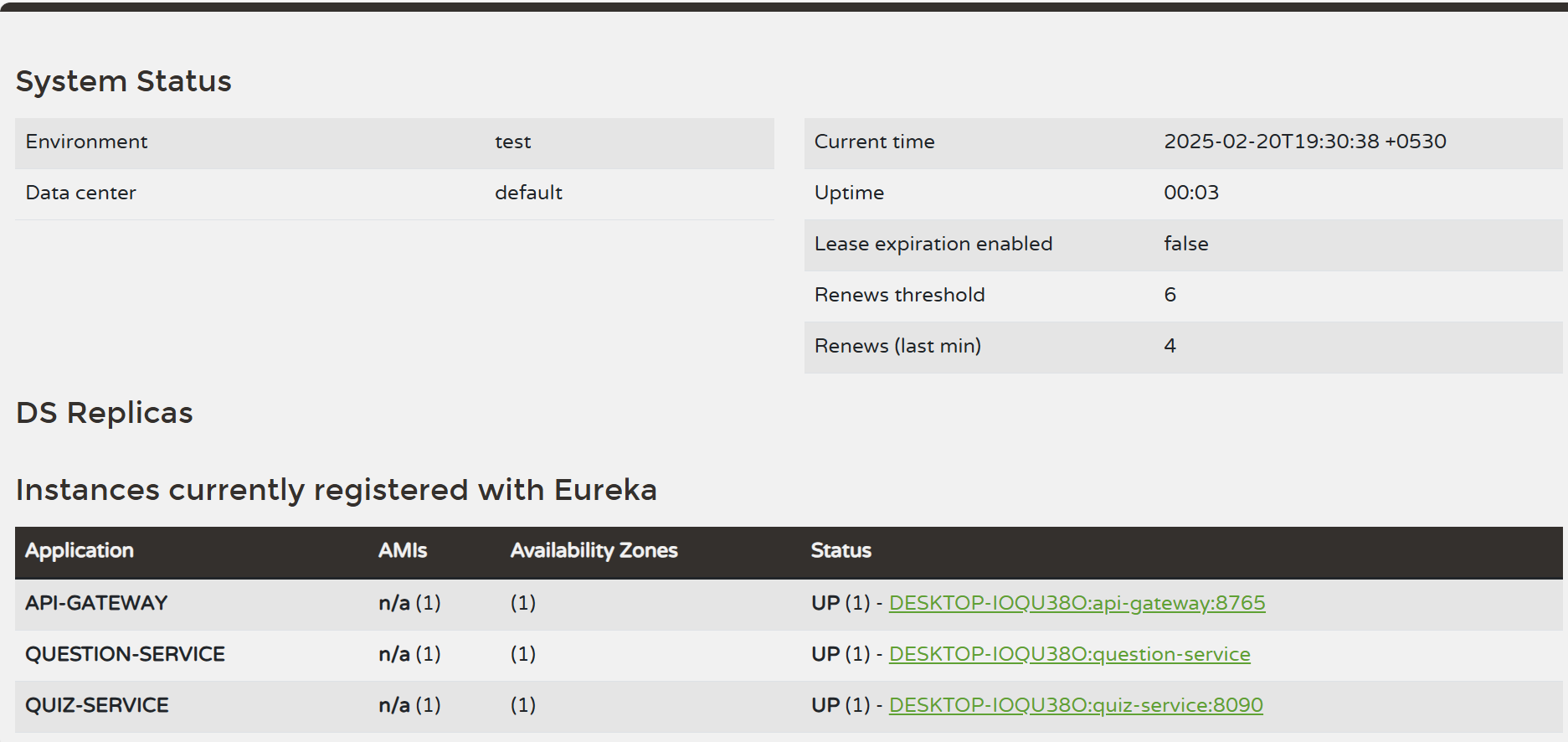
eureka.instance.hostname=localhost

eureka.client.fetch-registry=false

eureka.client.register-with-eureka=false







**Working with Feign**

**mport** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableFeignClients

**public** **class** QuizServiceApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(QuizServiceApplication.**class**, args);

}

}

**Feign Interface**

A Feign client is a Java-based tool that simplifies the process of creating HTTP clients to interact with web services by letting you define an interface with annotations that map directly to REST API calls, essentially allowing you to write code that looks like you're calling local methods while Feign handles the underlying network communication and request/response handling behind the scenes; it's particularly useful in microservices architectures where you need to easily call other services

Without Feign, calling another service requires writing manual REST API calls.

With Feign, it’s as simple as defining an interface!

**Without Feign (Using RestTemplate)**

RestTemplate restTemplate = new RestTemplate();

ResponseEntity<User> response = restTemplate.getForEntity("http://USER-SERVICE/users/1", User.class);

User user = response.getBody();

**With Feign Client (Simpler & Cleaner)**

@FeignClient(name = "USER-SERVICE")

public interface UserClient {

    @GetMapping("/users/{id}")

    User getUserById(@PathVariable("id") Long id);

}

Feign automatically finds `USER-SERVICE` and calls the correct API.

**import** com.telusko.quizservice.model.QuestionWrapper;

**import** com.telusko.quizservice.model.Response;

**import** org.springframework.cloud.openfeign.FeignClient;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PostMapping;

**import** org.springframework.web.bind.annotation.RequestBody;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** java.util.List;

@FeignClient("QUESTION-SERVICE")

**public** **interface** QuizInterface {

@GetMapping("question/generate")

**public** ResponseEntity<List<Integer>> getQuestionsForQuiz

(@RequestParam String categoryName, @RequestParam Integer numQuestions );

@PostMapping("question/getQuestions")

**public** ResponseEntity<List<QuestionWrapper>> getQuestionsFromId(@RequestBody List<Integer> questionIds);

@PostMapping("question/getScore")

**public** ResponseEntity<Integer> getScore(@RequestBody List<Response> responses);

}

**Service class**

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.stereotype.Service;

**import** java.util.List;

@Service

**public** **class** QuizService {

@Autowired

QuizDao quizDao;

@Autowired

QuizInterface quizInterface;

**public** ResponseEntity<String> createQuiz(String category, **int** numQ, String title) {

List<Integer> questions = quizInterface.getQuestionsForQuiz(category, numQ).getBody();

Quiz quiz = **new** Quiz();

quiz.setTitle(title);

quiz.setQuestionIds(questions);

quizDao.save(quiz);

**return** **new** ResponseEntity<>("Success", HttpStatus.***CREATED***);

}

**public** ResponseEntity<List<QuestionWrapper>> getQuizQuestions(Integer id) {

Quiz quiz = quizDao.findById(id).get();

List<Integer> questionIds = quiz.getQuestionIds();

ResponseEntity<List<QuestionWrapper>> questions = quizInterface.getQuestionsFromId(questionIds);

**return** questions;

}

**public** ResponseEntity<Integer> calculateResult(Integer id, List<Response> responses) {

ResponseEntity<Integer> score = quizInterface.getScore(responses);

**return** score;

}

}

**API GATEWAY**

An API Gateway is a crucial component in a microservices architecture, acting as a single entry point for client requests and managing communication between clients and multiple microservices. It simplifies complexity, enhances security, and improves performance, making it indispensable for building scalable and resilient systems.

Instead of exposing each microservice separately, we use an API Gateway to manage all traffic.

**Why API Gateway?**

✔ Centralized authentication & security (JWT, OAuth2)

✔ Load balancing & request routing

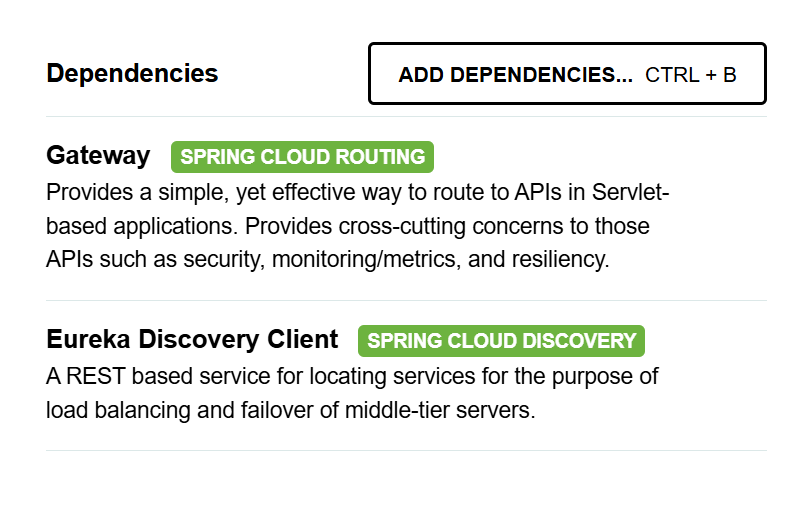
✔ Single entry point for all microservices

🔹 **Without API Gateway:**

* Clients must call services directly (e.g. `http://user-service:8081/users`)
* This exposes internal service details.

🔹 **With API Gateway:**

* Clients only call one URL (`http://api-gateway/users`).
* The API Gateway forwards the request to the right microservice.



spring.application.name=api-gateway

server.port=8765

spring.cloud.gateway.discovery.locator.enabled=true

spring.cloud.gateway.discovery.locator.lower-case-service-id=true

GitHub - alchimya/micro-node-service-registry-lib: An utility library ...

**How Everything Works Together:**

* Eureka Server registers all microservices.
* Microservices (Eureka Clients) register themselves with Eureka.
* Feign Client calls microservices dynamically via Eureka.
* API Gateway handles all client requests and forwards them to the correct microservice.

**Summary**

**✔ Feign Client simplifies inter-service communication.**

**✔ Eureka Server enables automatic service discovery.**

**✔ API Gateway provides a single entry point for all services.**