UỶ BAN NHÂN DÂN THÀNH PHỐ HỒ CHÍ MINH

**TRƯỜNG ĐẠI HỌC SÀI GÒN**

A blue circle with text

AI-generated content may be incorrect.

**BÁO CÁO GIỮA KỲ**

**CÁC CÔNG NGHỆ LẬP TRÌNH HIỆN ĐẠI**

**Sinh viên thực hiện:**

**Đoàn Minh Khôi – DCT121C3 – 3121411106**

**Trần Minh Chiến – DCT121C3 -3121411028**

**Nguyễn Quang Hiếu – DCT121C3- 31214111..**

# 

Giảng viên phụ trách**: TS. Đỗ Như Tài**

**TP. Hồ Chí Minh, tháng 03 năm 2025**

Contents

[Chương I. Kiến trúc Microservice 2](#_Toc195999976)

[1. Service Discovery & Configuration 3](#_Toc195999977)

[2. API Gateway 3](#_Toc195999978)

[3. Circuit Breaker & Resilience 3](#_Toc195999979)

[Chương II. Docker. 4](#_Toc195999980)

**[1.](#_Toc195999981)** [Giới thiệu về Docker 4](#_Toc195999981)

**[2.](#_Toc195999982)** [Các khái niệm cơ bản 4](#_Toc195999982)

**[3.](#_Toc195999983)** [Viết Dockerfile cho ứng dụng Spring Boot 4](#_Toc195999983)

[4. Docker Compose – Quản lý nhiều container 5](#_Toc195999984)

[5. Docker Volume & Network 6](#_Toc195999985)

[6. Quản lý cấu hình trong Docker 6](#_Toc195999986)

[7. Đẩy image lên Docker Hub 6](#_Toc195999987)

[8. Kết hợp Docker với CI/CD 7](#_Toc195999988)

[Chương III. Java Spring Boots 7](#_Toc195999989)

[1. Giới thiệu Spring Boot 7](#_Toc195999990)

[2. Kiến trúc của ứng dụng Spring Boot 7](#_Toc195999991)

[3. Tạo REST API cơ bản 8](#_Toc195999992)

[4. Cấu hình với application.yml 9](#_Toc195999993)

[5. Tích hợp với Spring Cloud 9](#_Toc195999994)

[6. Spring Boot Actuator & Monitoring 10](#_Toc195999995)

[7. Triển khai Spring Boot với Docker 10](#_Toc195999996)

[8. Unit Test và Integration Test 10](#_Toc195999997)

# Chương I. Kiến trúc Microservice

Kiến trúc Microservice là một mô hình phát triển phần mềm hiện đại, trong đó ứng dụng được chia thành các dịch vụ nhỏ, độc lập với nhau, có thể triển khai, mở rộng và duy trì riêng biệt. Để các microservice hoạt động hiệu quả và tin cậy trong môi trường phân tán, cần triển khai các thành phần hỗ trợ như Service Discovery, Cấu hình tập trung, API Gateway và Cơ chế phục hồi (Resilience).

## Service Discovery & Configuration

**Service Discovery (Khám phá dịch vụ)**

Trong môi trường Microservice, các dịch vụ thường được triển khai động (có thể thay đổi IP/host theo thời gian). Để các dịch vụ có thể giao tiếp với nhau một cách linh hoạt mà không phụ thuộc vào địa chỉ cứng (hardcoded), cần sử dụng **Service Discovery**:

* **Spring Cloud Netflix Eureka**:  
  Eureka là một Service Registry do Netflix phát triển. Các microservice có thể đăng ký (register) và tra cứu (lookup) thông tin dịch vụ tại Eureka Server. Spring Cloud cung cấp tích hợp chặt chẽ với Eureka, giúp việc kết nối và định vị dịch vụ dễ dàng thông qua annotation như @EnableEurekaClient.
* **Kubernetes Native Service Discovery**:  
  Trong môi trường Kubernetes, việc khám phá dịch vụ được hỗ trợ một cách tự động thông qua DNS và tên dịch vụ (Service name). Khi một Pod hoặc Service được triển khai, Kubernetes tự động tạo các DNS entries để các service khác có thể truy cập dễ dàng, không cần cấu hình thêm như với Eureka.

**Configuration Management (Quản lý cấu hình)**

Việc quản lý cấu hình tập trung là rất quan trọng để đảm bảo tính nhất quán và dễ kiểm soát cấu hình cho từng microservice, đặc biệt khi thay đổi môi trường (dev, staging, production):

* **Spring Cloud Config Server**:  
  Cho phép quản lý cấu hình tập trung từ một kho lưu trữ (thường là Git). Các service có thể lấy cấu hình lúc khởi động hoặc tự động cập nhật khi có thay đổi. Điều này giúp giảm thiểu rủi ro khi chỉnh sửa cấu hình thủ công trên từng service.
* **Kubernetes ConfigMaps & Secrets**:  
  Kubernetes hỗ trợ quản lý cấu hình và thông tin nhạy cảm (password, token...) thông qua **ConfigMaps** và **Secrets**. Các giá trị này có thể được mount vào container dưới dạng biến môi trường hoặc file cấu hình, giúp dễ dàng cập nhật mà không cần rebuild ứng dụng.

1. API Gateway

API Gateway là điểm vào (entry point) duy nhất cho các client (web, mobile...) khi tương tác với hệ thống microservice. Nó đóng vai trò điều phối các request đến đúng service, thực hiện xác thực, logging, giới hạn tốc độ (rate limiting) và chuyển đổi định dạng dữ liệu nếu cần.

* **Spring Cloud Gateway**:  
  Là API Gateway chính thức của hệ sinh thái Spring, thay thế cho Zuul. Spring Cloud Gateway hỗ trợ routing thông minh, load balancing, filter tùy chỉnh, và tích hợp tốt với Eureka và Config Server. Việc cấu hình đơn giản thông qua YAML hoặc Java DSL.
* **Kong**:  
  Là một API Gateway mã nguồn mở rất mạnh mẽ, hỗ trợ plugin phong phú như xác thực OAuth2, rate limiting, logging, monitoring,... Kong thường được triển khai trong môi trường Kubernetes kết hợp với database backend (PostgreSQL hoặc Cassandra).
* **Nginx Ingress Controller**:  
  Dành riêng cho môi trường Kubernetes, Ingress Controller sử dụng Nginx để định tuyến các request HTTP/HTTPS đến đúng service backend. Nó cung cấp khả năng tùy chỉnh cao, dễ tích hợp SSL, rewrite URL và nhiều tính năng gateway truyền thống khác.

## Circuit Breaker & Resilience

Trong kiến trúc phân tán, khi một service bị lỗi hoặc không phản hồi, cần có cơ chế ngắt mạch (circuit breaker) để tránh gây sụp đổ toàn bộ hệ thống do phản ứng dây chuyền. Đồng thời, cần bổ sung khả năng tự phục hồi (resilience) và retry thông minh.

* **Resilience4j**:  
  Là thư viện nhẹ, hiện đại, được thiết kế thay thế cho Hystrix (đã bị ngừng phát triển). Resilience4j hỗ trợ nhiều tính năng như:
  + **Circuit Breaker**: Ngắt mạch khi service liên tục thất bại.
  + **Retry**: Tự động thử lại khi gặp lỗi tạm thời.
  + **Rate Limiter**: Giới hạn số lượng request trong khoảng thời gian.
  + **Bulkhead**: Cô lập lỗi theo thread-pool hoặc semaphore để tránh lan rộng.

Resilience4j tích hợp tốt với Spring Boot thông qua các starter như spring-cloud-starter-circuitbreaker-resilience4j, và hỗ trợ cấu hình thông qua application.yml.

# Chương II. Docker.

1. Giới thiệu về Docker

Docker là nền tảng container hóa (containerization platform) giúp đóng gói ứng dụng và các phụ thuộc vào trong một container độc lập, nhẹ, có thể chạy đồng nhất trên nhiều môi trường (local, staging, production). Docker đặc biệt phù hợp trong kiến trúc Microservice khi mỗi service có thể được đóng gói và triển khai riêng biệt.

**Ưu điểm của Docker:**

* **Đóng gói độc lập**: Mỗi ứng dụng chạy trong container riêng, không ảnh hưởng đến nhau.
* **Tái sử dụng và chia sẻ dễ dàng**: Có thể chia sẻ container image thông qua Docker Hub.
* **Triển khai nhanh chóng**: Khởi động container nhanh hơn so với VM.
* **Nhất quán môi trường phát triển – triển khai**: "Chạy được ở máy tôi" cũng sẽ chạy được ở production.

1. Các khái niệm cơ bản

* **Docker Image**: Ảnh mẫu (template) dùng để tạo container. Image bao gồm ứng dụng và các dependency.
* **Docker Container**: Thực thể chạy của một image.
* **Dockerfile**: Tập tin định nghĩa cách build ra một image.
* **Docker Hub**: Kho lưu trữ image cộng đồng.
* **Docker Compose**: Công cụ giúp định nghĩa và chạy nhiều container cùng lúc (thường dùng cho hệ thống microservice).

## Viết Dockerfile cho ứng dụng Spring Boot

1. # Sử dụng JDK base image

2. FROM openjdk:17-jdk-slim

3.

4. # Thêm file jar vào image

5. COPY target/myapp.jar app.jar

6.

7. # Khai báo port sẽ expose

8. EXPOSE 8080

9.

10. # Lệnh khởi chạy ứng dụng

11. ENTRYPOINT ["java", "-jar", "/app.jar"]

Sau đó build image:

1. docker build -t myapp .

Chạy container:

1. docker run -p 8080:8080 myapp

## 4. Docker Compose – Quản lý nhiều container

Docker Compose cho phép định nghĩa cấu hình nhiều container (microservices) chỉ trong một file docker-compose.yml.

Ví dụ hệ thống gồm:

* API Gateway
* Eureka Server
* 2 service (User, Product)
* PostgreSQL database

1. version: '3.8'

2. services:

3. eurekaserver:

4. image: eurekaserver

5. ports:

6. - "8761:8761"

7.

8. apigateway:

9. image: apigateway

10. ports:

11. - "8080:8080"

12. depends\_on:

13. - eurekaserver

14.

15. userservice:

16. image: userservice

17. ports:

18. - "8081:8081"

19. depends\_on:

20. - eurekaserver

21. - db

22.

23. productservice:

24. image: productservice

25. ports:

26. - "8082:8082"

27. depends\_on:

28. - eurekaserver

29. - db

30.

31. db:

32. image: postgres:14

33. environment:

34. POSTGRES\_USER: user

35. POSTGRES\_PASSWORD: password

36. POSTGRES\_DB: mydb

37. ports:

38. - "5432:5432"

Chạy toàn bộ hệ thống:

1. docker-compose up --build

## 5. Docker Volume & Network

Volumes dùng để lưu trữ dữ liệu vĩnh viễn ngoài container.

1. docker volume create postgres\_data

Networks giúp các container giao tiếp trong cùng một môi trường:

1. docker network create my-network

Trong Docker Compose, network được tạo tự động và các service có thể truy cập nhau qua tên service.

## 6. Quản lý cấu hình trong Docker

* **Sử dụng .env file**: Cho phép đưa các biến môi trường vào ứng dụng.
* **Kết hợp với Spring Boot**: Spring hỗ trợ lấy config từ biến môi trường (ENV), thích hợp khi chạy trong container.

Ví dụ Docker Compose:

1. environment:

2. - SPRING\_DATASOURCE\_URL=jdbc:postgresql://db:5432/mydb

3. - SPRING\_DATASOURCE\_USERNAME=user

4. - SPRING\_DATASOURCE\_PASSWORD=password

5.

## 7. Đẩy image lên Docker Hub

1. docker tag myapp username/myapp:latest

2. docker push username/myapp:latest

Sau đó có thể kéo và chạy từ bất kỳ server nào:

1. docker pull username/myapp

2. docker run -p 8080:8080 username/myapp

## 8. Kết hợp Docker với CI/CD

Docker là thành phần không thể thiếu trong pipeline CI/CD hiện đại. Trong quá trình build, có thể:

* Build image
* Push lên Docker Registry
* Deploy lên server hoặc Kubernetes

Các công cụ thường dùng:

* Jenkins
* GitHub Actions
* GitLab CI/CD

# Chương III. Java Spring Boots

## 1. Giới thiệu Spring Boot

**Spring Boot** là một phần mở rộng của framework Spring, giúp đơn giản hóa quá trình phát triển ứng dụng Java bằng cách cung cấp:

* Cấu hình mặc định (convention over configuration)
* Tích hợp sẵn với nhiều công nghệ phổ biến
* Khởi tạo và triển khai nhanh chóng (có thể chạy như một ứng dụng độc lập)

Spring Boot là lựa chọn lý tưởng cho kiến trúc Microservice vì:

* Hỗ trợ REST API dễ dàng
* Tích hợp tốt với Docker, Spring Cloud, Eureka, Config Server
* Có hệ sinh thái đa dạng: Spring Data JPA, Security, Actuator, etc.

## 2. Kiến trúc của ứng dụng Spring Boot

Một ứng dụng Spring Boot thông thường bao gồm các thành phần:

* **Controller**: Xử lý request từ phía client
* **Service**: Chứa logic xử lý nghiệp vụ
* **Repository**: Giao tiếp với CSDL (qua Spring Data JPA)
* **Entity/Model**: Đại diện cho các bảng dữ liệu
* **Configuration**: Các lớp cấu hình, beans

Cấu trúc thư mục ví dụ:

1. src/main/java/com/example/demo

2. │

3. ├── controller

4. ├── service

5. ├── repository

6. ├── model

7. ├── config

8. └── DemoApplication.java

## 3. Tạo REST API cơ bản

**Ví dụ**: API quản lý sản phẩm

**Entity**

1. @Entity

2. public class Product {

3. @Id @GeneratedValue

4. private Long id;

5. private String name;

6. private double price;

7. // getters/setters

8. }

9.

Repository

1. public interface ProductRepository extends JpaRepository<Product, Long> {}

Service

1. @Service

2. public class ProductService {

3. @Autowired

4. private ProductRepository repo;

5.

6. public List<Product> getAll() {

7. return repo.findAll();

8. }

9.

10. public Product save(Product p) {

11. return repo.save(p);

12. }

13. }

Controller

1. @RestController

2. @RequestMapping("/api/products")

3. public class ProductController {

4. @Autowired

5. private ProductService service;

6.

7. @GetMapping

8. public List<Product> getAll() {

9. return service.getAll();

10. }

11.

12. @PostMapping

13. public Product create(@RequestBody Product p) {

14. return service.save(p);

15. }

16. }

## 4. Cấu hình với application.yml

1. server:

2. port: 8081

3.

4. spring:

5. datasource:

6. url: jdbc:postgresql://localhost:5432/mydb

7. username: user

8. password: password

9. jpa:

10. hibernate:

11. ddl-auto: update

12. show-sql: true

13.

## 5. Tích hợp với Spring Cloud

Spring Boot kết hợp chặt chẽ với các module trong Spring Cloud để triển khai kiến trúc microservice như:

* **Spring Cloud Netflix Eureka**: Đăng ký và phát hiện service
* **Spring Cloud Config**: Cấu hình tập trung
* **Spring Cloud Gateway**: Gateway API
* **Spring Cloud Circuit Breaker (Resilience4j)**: Tăng độ bền vững cho hệ thống

Ví dụ khai báo client Eureka:

1. eureka:

2. client:

3. serviceUrl:

4. defaultZone: http://localhost:8761/eureka/

Annotation:

1. @EnableEurekaClient

2. @SpringBootApplication

3. public class ProductServiceApplication {}

## 6. Spring Boot Actuator & Monitoring

**Spring Boot Actuator** cung cấp các endpoint giúp giám sát và quản lý ứng dụng:

1. management:

2. endpoints:

3. web:

4. exposure:

5. include: "\*"

6.

Sau đó có thể truy cập các endpoint như:

* /actuator/health
* /actuator/metrics
* /actuator/info

Kết hợp với Prometheus + Grafana để vẽ biểu đồ giám sát.

## 7. Triển khai Spring Boot với Docker

**Dockerfile**

1. FROM openjdk:17-jdk-slim

2. COPY target/product-service.jar app.jar

3. EXPOSE 8081

4. ENTRYPOINT ["java", "-jar", "/app.jar"]

**Build & Run**

1. docker build -t product-service .

2. docker run -p 8081:8081 product-service

## 8. Unit Test và Integration Test

Spring Boot hỗ trợ viết test dễ dàng:

**Unit Test (với JUnit + Mockito)**

1. @RunWith(MockitoJUnitRunner.class)

2. public class ProductServiceTest {

3. @InjectMocks private ProductService service;

4. @Mock private ProductRepository repo;

5.

6. @Test

7. public void testGetAll() {

8. when(repo.findAll()).thenReturn(List.of(new Product()));

9. assertEquals(1, service.getAll().size());

10. }

11. }

**Integration Test (với @SpringBootTest)**

1. @SpringBootTest

2. @AutoConfigureMockMvc

3. public class ProductControllerTest {

4. @Autowired private MockMvc mockMvc;

5.

6. @Test

7. public void testGetAll() throws Exception {

8. mockMvc.perform(get("/api/products"))

9. .andExpect(status().isOk());

10. }

11. }