# **I. TỔNG QUAN MÔN HỌC**

1. **Tìm hiểu các kiểu kiến trúc chính**

Kiến trúc hệ thống đóng vai trò quan trọng như nền móng của một tòa nhà trong lĩnh vực phát triển  phần mềm. Kiến trúc không chỉ quyết định cách các thành phần phần mềm tương tác mà còn ảnh  hưởng trực tiếp đến khả năng mở rộng, bảo trì và hiệu suất của hệ thống.

**Các kiểu kiến trúc chính**:

* **Monolithic Architecture**: Tất cả các chức năng của ứng dụng nằm trong một khối duy  nhất, dễ triển khai nhưng khó mở rộng.
* **Modular Monolithic Architecture**: Phân tách thành các mô-đun độc lập bên trong một  khối ứng dụng, giúp tăng tính tổ chức và bảo trì.
* **Microservices Architecture**: Chia ứng dụng thành các dịch vụ nhỏ, độc lập, dễ mở rộng và duy trì nhưng phức tạp hơn khi triển khai.

**Nguyên tắc thiết kế**:

* **KISS (Keep It Simple, Stupid)**: Thiết kế càng đơn giản càng tốt.
* **YAGNI (You Aren’t Gonna Need It)**: Không thêm các tính năng không cần thiết.
* **Separation of Concerns (SoC)**: Phân tách trách nhiệm giữa các thành phần.
* **SOLID Principles**: Giúp thiết kế có tính mở rộng, dễ bảo trì.

1. **Microservices Architecture**

Nhu cầu xây dựng các hệ thống linh hoạt, dễ mở rộng và duy trì đã đưa Microservices Architecture trở thành một trong những xu hướng thiết kế phổ biến nhất. Không giống như kiến trúc monolithic truyền  thống, microservices phân chia ứng dụng thành các dịch vụ nhỏ, độc lập, giúp tối ưu hóa hiệu năng,  giảm độ phức tạp trong phát triển và triển khai. Các mô hình giao tiếp giữa các dịch vụ như  RESTful API, gRPC, và WebSocket, đồng thời khám phá những pattern phổ biến trong giao tiếp và  triển khai microservices.

**Đặc điểm chính:**

* Dịch vụ độc lập, triển khai linh hoạt, hỗ trợ công nghệ đa dạng.
* Khả năng mở rộng theo chiều ngang, phù hợp với khối lượng lớn yêu cầu người dùng.
* Cách ly lỗi, dễ phục hồi trong các tình huống thất bại.

**Giao tiếp giữa các dịch vụ:**

* HTTP-based RESTful API: Phổ biến, dễ triển khai.
* gRPC: Giao tiếp hiệu suất cao, dùng mã hóa nhị phân.
* WebSocket: Hỗ trợ giao tiếp thời gian thực, hai chiều.

**Các pattern giao tiếp:**

* Gateway Routing Pattern: Định tuyến các yêu cầu qua API Gateway.
* Gateway Aggregation Pattern: Tổng hợp dữ liệu từ nhiều dịch vụ.
* Backends for Frontends (BFF): Tối ưu hóa giao diện với API dành riêng.

**3. Quản lý dữ liệu trong microservices**

Quản lý dữ liệu trong microservices phức tạp do mỗi service có cơ sở dữ liệu riêng, gây thách thức về đồng bộ và nhất quán. Các kỹ thuật quan trọng gồm **Database-per-Service**, **Data Partitioning** (phân mảnh ngang, dọc, theo chức năng) để tối ưu hiệu suất. Ngoài ra, các pattern như **CQRS, Event Sourcing,** và **Saga** giúp xử lý truy vấn, ghi dữ liệu, và giao dịch phân tán hiệu quả.

**Database-per-Service Pattern**: Mỗi dịch vụ sở hữu một cơ sở dữ liệu riêng, đảm bảo tính độc  lập và cách ly dữ liệu.

**Data Partitioning**:

* Horizontal Partitioning: Phân chia theo hàng.
* Vertical Partitioning: Phân chia theo cột.
* Functional Partitioning: Phân chia theo chức năng.

**CQRS (Command Query Responsibility Segregation)**: Phân tách giữa xử lý truy vấn (query) và xử lý lệnh (command), tăng hiệu suất.

**Event Sourcing Pattern**:

* Lưu trữ toàn bộ các sự kiện thay đổi trạng thái, không lưu trạng thái cuối.
* Hỗ trợ phục hồi và tái hiện trạng thái hệ thống.

**Distributed Transactions**:

* Saga Pattern: Chia giao dịch lớn thành các bước nhỏ, dễ quản lý và phục hồi khi thất  bại.
* Transactional Outbox Pattern: Đảm bảo tính nhất quán khi ghi dữ liệu ra cơ sở dữ liệu.
* Change Data Capture (CDC): Ghi nhận các thay đổi dữ liệu để đồng bộ hóa.

**4. Resilience và triển khai hệ thống:**

Trong hệ thống microservices, lỗi có thể gây hiệu ứng domino, nên khả năng chịu lỗi (resilience) và triển khai hiệu quả là yếu tố quan trọng. Các resilience patterns như Retry, Circuit Breaker, Bulkhead, Timeout giúp giảm tác động của lỗi. Công cụ giám sát như Prometheus, Grafana, Elastic Stack hỗ trợ theo dõi hệ thống. Về triển khai, các chiến lược Blue-Green, Rolling, Canary Deployment giúp cập nhật không gián đoạn. Cuối cùng, Docker và Kubernetes đóng vai trò quan trọng trong tự động hóa và quản lý triển khai.

**Resilience Patterns**:

* Retry: Thử lại khi gặp lỗi tạm thời.
* Circuit Breaker: Ngắt kết nối nếu phát hiện lỗi thường xuyên.
* Bulkhead: Phân vùng tài nguyên để giảm thiểu ảnh hưởng của lỗi.
* Timeout: Giới hạn thời gian chờ phản hồi.

**Giám sát và quan sát hệ thống**:

* Prometheus: Thu thập và giám sát chỉ số hiệu năng.
* Grafana: Trực quan hóa dữ liệu giám sát.
* Elastic Stack: Phân tích nhật ký hệ thống.

**Chiến lược triển khai**:

* Blue-Green Deployment: Giảm thời gian chết bằng cách triển khai vào môi trường  mới.
* Rolling Deployment: Cập nhật lần lượt từng phần của hệ thống.
* Canary Deployment: Triển khai thử nghiệm trên một phần nhỏ trước khi áp dụng toàn  hệ thống.

**5. Các bài toán thực tế và giải pháp**

Khi triển khai microservices, các vấn đề thực tế như giao tiếp giữa dịch vụ, tối ưu hiệu suất, xử lý dữ liệu lớn, và giao dịch phân tán thường gặp. Các giải pháp gồm:

* Giao tiếp giữa dịch vụ: Giảm độ trễ với Asynchronous Messaging và Service Aggregator.
* Hiệu suất hệ thống: Tăng tốc với caching phân tán, sharding, và tối ưu hóa truy vấn.
* Dữ liệu lớn: Quản lý lượng sự kiện lớn bằng event-driven architecture và stream processing.
* Giao dịch phân tán: Đảm bảo nhất quán với Saga Pattern, CQRS, và Event Sourcing.

**Giao tiếp giữa các dịch vụ**:

* Vấn đề: Giao tiếp đồng bộ giữa các dịch vụ gây ra độ trễ cao.
* Giải pháp: Sử dụng giao tiếp bất đồng bộ với Event-driven Architecture.

**Hiệu suất hệ thống**:

* Vấn đề: Nhiều truy vấn tới cơ sở dữ liệu làm giảm hiệu suất.
* Giải pháp: Dùng caching phân tán, sharding cơ sở dữ liệu.

**Quản lý dữ liệu lớn**:

* Vấn đề: Xử lý hàng triệu sự kiện mỗi giây.
* Giải pháp: Sử dụng kiến trúc Event-driven với stream processing.

**Quản lý giao dịch phân tán**:

* Vấn đề: Đảm bảo tính nhất quán trong giao dịch trên nhiều dịch vụ.
* Giải pháp: Áp dụng Saga Pattern và CQRS để quản lý giao dịch.

**6. Tư duy thiết kế**

Trong phát triển phần mềm, **tư duy thiết kế đúng đắn** giúp hệ thống **mở rộng, thích nghi với tương lai** thay vì chỉ chạy theo công nghệ mới. Các nguyên tắc quan trọng gồm:

* **Tránh over-engineering:** Không dùng công nghệ phức tạp nếu không mang lại giá trị thực tế.
* **Dựa trên yêu cầu thực tế:** Quyết định thiết kế cần xem xét số lượng người dùng, thời gian đáp ứng, rủi ro và khả năng mở rộng.
* **Hướng tới mục tiêu dài hạn:** Giải quyết vấn đề trước mắt nhưng vẫn đảm bảo tính phát triển bền vững.

**Thiết kế dựa trên yêu cầu kinh doanh**:

* Tránh thiết kế quá phức tạp (over-engineering).
* Ưu tiên các tính năng đáp ứng trực tiếp yêu cầu kinh doanh.

**Định hướng mở rộng**:

* Thiết kế ban đầu cần linh hoạt, dễ mở rộng.
* Xây dựng hệ thống đáp ứng các chỉ tiêu như số lượng người dùng đồng thời, thời gian  đáp ứng yêu cầu.

# **II. TÌM HIỂU DOCKER VÀ WSL2**

**1. Docker là gì?**

Docker là một nền tảng mã nguồn mở cho phép đóng gói, phân phối và chạy các ứng dụng dưới dạng các container độc lập.

* Một container chứa tất cả các thành phần cần thiết để chạy một ứng dụng (mã nguồn, thư viện, môi trường runtime...).
* Khác với máy ảo (VM), Docker sử dụng chung nhân hệ điều hành, giúp tiết kiệm tài nguyên và chạy nhanh hơn.
* Container giúp triển khai ứng dụng dễ dàng, nhanh chóng, đồng nhất trên mọi hệ thống.

**2. Các ưu điểm của Docker**

* Di động (Portable): Chạy được trên Windows, Linux, macOS mà không lo lỗi môi trường.
* Nhẹ hơn máy ảo: Không cần chạy hệ điều hành riêng, giúp tiết kiệm RAM, CPU.
* Triển khai nhanh chóng: Dễ dàng đóng gói ứng dụng và chạy trên mọi nền tảng.
* **Dễ quản lý**: Có thể chạy nhiều container cùng lúc mà không bị xung đột.

**3. Các thành phần chính của Docker**

* Docker Engine: Chạy các container trên máy tính của bạn.
* Docker Image: Ảnh (image) là bản đóng gói sẵn của ứng dụng và các thư viện cần thiết.
* Docker Container: Là một phiên bản đang chạy của image, tương tự như một process trên máy tính.
* Docker Hub: Kho lưu trữ các image có sẵn để sử dụng hoặc chia sẻ với người khác.
* Dockerfile: File mô tả cách xây dựng Image.

**4. Minh họa**

**Ví dụ về Dockerfile (Tạo dự án Nodejs)**

**Tạo file Package.json:**

{  
 "name": "docker-node-app",  
 "version": "1.0.0",  
 "main": "index.js",  
 "scripts": {  
 "start": "node index.js"  
 },  
 "dependencies": {  
 "express": "^4.17.1"  
 }  
}

**Tạo file index.js**

*const* express = *require*('express');  
*const* app = express();  
*const* port = *process*.*env*.PORT || 3000;  
  
app.get('/', (req, res) *=>* {  
 res.send('Hello World!');  
});  
  
app.listen(port, () *=>* {  
 *console*.*log*(`App is running on port ${port}`);  
});

**Tạo file Dockerfile**

# Sử dụng image cơ sở   
FROM node:14   
  
# Thiết lập thư mục làm việc  
WORKDIR /app  
  
# Sao chép file package.json vào thư mục làm việc  
COPY package.json ./   
  
# Cài đặt các dependencies  
RUN npm install   
  
# Sao chép toàn bộ mã nguồn vào thư mục làm việc  
*COPY* . .   
  
# Thiết lập biến môi trường  
ENV PORT=3000  
  
# Mở cổng 3000  
EXPOSE 3000   
  
# Chạy ứng dụng  
*CMD* ["node", "index.js"]

**Build Dockerfile**

docker build -t docker-node-app .  
  
docker run -p 3000:3000 docker-node-app