# Design pattern là gì?

Design pattern là các giải pháp tổng thể đã được tối ưu hóa, được tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm mà chúng ta thường gặp phải hàng ngày. Đây là tập các giải pháp đã được suy nghĩ, đã giải quyết trong tình huống cụ thể.

Là một kỹ thuật trong lập trình hướng đối tượng, không phải ngôn ngữ nào cả, design pattern là một kỹ thuật lập trình. Có thể được thực hiện ở phần lớn các ngôn ngữ lập trình, design pattern là một khái niệm quan trọng và mọi developer đều cần phải biết, nó cung cấp cho bạn các “mẫu thiết kế”, giải pháp để giải quyêt các vấn đề chung mà thường gặp trong lập trình. Các vấn đề mà ở đó, bạn có thể tự nghĩa ra giải pháp nhưng co thể chưa được tôi ưu.

Design pattern giúp bạn giải quyết vấn đề một cách tối ưu, cung cấp cho bạn các giải pháp trong lập trình OOP. Tuy nhiên, nếu bạn là người mới hay chưa rõ về OOP thì không nên tìm hiểu về no lúc này. Để tìm hiểu về design pattern, bạn cần có kiến thức vững chắc về lập trình, đặc biệt là OOP.

# Microservice architecture pattern

Bạn đang phát triển 1 ứng dụng enterprise phía server. Nó phải hỗ trợ nhiều loại client khác nhau bao gồm cả các browser desktop, mobile và các ứng dụng mobile. Ứng dụng cũng có thể chia sẻ API cho bên thứ 3 sử dụng. Nó cũng có thể tich hợp với các ứng dụng khác thông qua các web service khác hoặc 1 message broker. Ứng dụng sử lý các request (HTTP request và message) bằng việc thực hiện các logic nghiệp vụ; truy cập 1 database; trao đổi message với các hệ thống khác; và trả về 1 HTML/JSON/XML response. Có các thành phần tương ứng với các vùng chức năng khác nhau của ứng dụng.

Cần 1 kiến trức cấy trúc ứng dụng như 1 tập các service được kết nối lỏng lẻo.

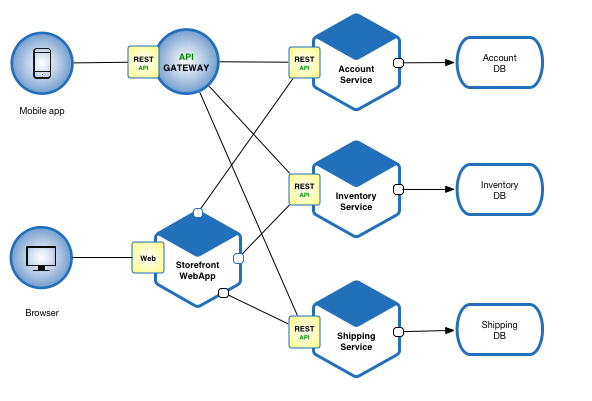
Kiến trúc microservice là 1 kiểu kiến trúc cấu trúc 1 ứng dụng như 1 tập hợp các service được tổ trức xoay quanh các khả năng nghiệp vụ. Nó cho phép phân phối/ triển khai liên tục cá ứng dụng lớn và phức tạp. Kiển trúc microservice có 1 số lợi ích khác bao gồm khả năng mở rộng và khả năng chịu lỗi tốt hơn. Nó giúp tổ chức dễ dàng thử nghiệm cacs công nghệ mới và phát triển nền tảng công nghệ của mình. Một service sẽ có các đặc tính sau:

* Có thể bảo trì và kiểm tra cao - cho phép phát triển và triển khai nhanh chóng và thường xuyên.
* Kết hợp lỏng lẻo với các dịch vụ khác – cho phép 1 nhóm làm việc độc lập trong phần lớn thời gian trên (các) service của họ mà không bị ảnh hưởng bởi các thay đổi đối với các service khác và không ảnh hưởng đến các service khác.
* Có thể triển khai độc lập – cho phép 1 nhóm triển khai service của họ mà không cần phải phối hợp với các nhóm khác.
* Có khả năng được phát triển bởi 1 nhóm nhỏ - cần thiết cho hiệu quả công việc cao bằng cách tránh high communication head giữa các nhóm lớn.

Các service giao tiếp với nhau bằng cách dung các giao thức đồng bộ của chúng như HTTP/REST hoặc bất đồng bộ như AMQP. Các service có thể được phát triển và triển khai độc lập với nhau. Mỗi service có database riêng để tách biệt với nhau. Tính nhất quán giữa các service được duy trì bằng cách sử dụng [Saga](#_Saga).

## Ví dụ: E-commerce application

Thử tưởng tượng rằng, bạn đang xây dựng 1 ứng dụng thương mại điện tử mà ở các order được tạo từ các khách hàng, xác minh inventory và số dư hiện có, và ship chúng đi. Ứng dụng này bao gồm 1 số thành phần bao gồm UI, nơi thực thi interface user, cùng với vài backend service để kiểm tra số dư, cập nhật inventory và ship các order. Ứng dụng bao gồm 1 tập hợp các service



## Lợi ích

Cho phép phân phối và triển khai liên tục các ứng dụng lớn, phức tạp.

* Cải thiện khả năng bảo trì - mỗi dịch vụ tương đối nhỏ nên dễ hiểu và dễ thay đổi hơn
* Khả năng kiểm tra tốt hơn - các dịch vụ nhỏ hơn và nhanh hơn để kiểm tra
* Khả năng triển khai tốt hơn - các dịch vụ có thể được triển khai độc lập
* Nó cho phép bạn tổ chức nỗ lực phát triển xung quanh nhiều nhóm tự trị. Mỗi nhóm (được gọi là hai pizza) sở hữu và chịu trách nhiệm cho một hoặc nhiều dịch vụ. Mỗi nhóm có thể phát triển, thử nghiệm, triển khai và mở rộng các dịch vụ của mình một cách độc lập với tất cả các nhóm khác.

Mỗi microservice tương đối nhỏ:

* Nhà phát triển dễ hiểu hơn
* IDE nhanh hơn giúp các nhà phát triển làm việc hiệu quả hơn
* Ứng dụng khởi động nhanh hơn, giúp các nhà phát triển làm việc hiệu quả hơn và tăng tốc độ triển khai

Cải thiện khả năng cách ly lỗi. Ví dụ, nếu có rò rỉ bộ nhớ trong một dịch vụ thì chỉ dịch vụ đó bị ảnh hưởng. Các dịch vụ khác sẽ tiếp tục xử lý các yêu cầu. Trong khi đó, một thành phần hoạt động sai của một kiến ​​trúc nguyên khối có thể làm hỏng toàn bộ hệ thống.

Loại bỏ bất kỳ cam kết lâu dài nào đối với hệ thống công nghệ. Khi phát triển một dịch vụ mới, bạn có thể chọn một công nghệ mới. Tương tự, khi thực hiện các thay đổi lớn đối với một dịch vụ hiện có, bạn có thể viết lại nó bằng cách sử dụng công nghệ mới.

## Hạn chế

Các developer phải đối mặt với sự phức tạp kèm theo của việc tạo 1 hệ thống phân tán:

* + Các developer phải thực thi cơ chế giao tiếp giữ các service và đối phó với từng thất bại.
  + Việc thực hiện các request đi qua nhiều service trở lên khó hơn.
  + Việc tra giao tiếp giữa các service trở lên khó hơn
  + Thực thi các request trải dài qua các service yêu cầu sự phối hợp cẩn thận giữa các team
  + Các công cụ / IDE dành cho nhà phát triển được định hướng vào việc xây dựng các ứng dụng nguyên khối và không cung cấp hỗ trợ rõ ràng cho việc phát triển các ứng dụng phân tán.

Triển khai trở lên phức tạp. Trong Production, cũng có sự phức tạp trong hoạt động của việc triển khai và quản lý một hệ thống bao gồm nhiều dịch vụ khác nhau.

# Mediator

Trong kĩ thuật phần mềm, mediator pattern xác định một đối tượng gói gọn cách một tập hợp các đối tượng tương tác. Pattern này thuộc nhóm pattern behavior do cách nó ó thể thay đổi hành vi đang chạy của chương trình.

Trong lập trình hướng đối tượng, chương trình thường bao gồm nhiều class, logic nghiệp vụ và tính toán được phân phối đến các class này. Tuy nhiên, khi nhiều class được them vào trong chương trình, đặc biệt là khi bảo trì và/hoặc tái cấu trúc, vấn đề giao tiếp giữa các class này có thể trở lên phức tạp. Điều này chương trình trở lên khó đọc và bảo trì. Hơn nữa, việc thay đổi chương trình trở nên khó khăn hơn, vì bất kì thay đổi nào cũng có thể ảnh hưởng đến code trong một số lớp khác.

Với mediator pattern, giao tiếp giữa các đối tượng được gói bên trong một đối tượng mediator. Các đối tượng không còn phải giao tiếp trực tiếp với nhau, thay vào đó là giao tiếp thông qua mediator. Điều này làm giảm sự phụ thược giữa các đối tượng giao tiếp.

## Lợi ích

* Tránh được sự kết nối chặt chẽ giữa một tập hợp các đối tượng tương tác.
* Có thể thay đổi sự tương tác giữa một tập hợp các đối tượng một cách độc lập.

Việc xác định một tập hợp các đối tượng giao biếp thông qua quá trình truy cập và cập nhật trực tiếp lẫn nhau là không linh hoạt vì nó kết hợp chặt chẽ các đối tượng với nhau và không thể thay đổi tương tác một cách độc lập với (không cần thay đổi) các đối tượng. Và nó ngăn không cho các đối tượng có thể tái sử dụng và khiến chúng khó kiểm thử.

*Các đối tượng được kết nối chặt chẽ khó để thực hiện, thay đổi, kiểm thử và tái sử dụng vì chúng tham chiêus và biết đến nhiều đối tượng khác*

## Mô tả

* Định nghĩa một đối tượng (mediator) riêng biệt đóng gói sự giao tiếp giữa một tập hợp các đối tượng.
* Các đối tượng ủy thác sự tương tác của chúng tới một đối tượng mediator thay vì giao tiếp trực tiếp với nhau.

Các đối tượng giao tiếp gián tiếp với nhau thông qua một đối tượng mediator điều khiển và phân phối sự giao tiếp.

Điều này tạo sự kết nối lỏng lẻo giữa các đối tượng. Chúng chỉ thâm chiếu và biết về đối tượng mediator của chúng và không có kiến thức rõ ràng về nhau.

*Trong C# có nugget package MediatR hỗ trợ việc triển khai mediator pattern.*

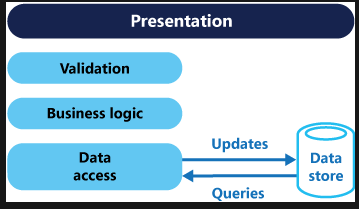
# CQRS

CQRS (command and query responsibility segregation) là một pattern phân tách các phương thức đọc và cập nhật cho một kho dữ liệu. Việc thực thi CQRS trong ứng dụng có thể tối đa hóa hiệu năng, khả năng mở rộng và bảo mật. Tính linh hoạt được tạo ra bằng cách chuyển sang CQRS cho phép một hệ thống phát triển tốt hơn theo thời gian và hạn chế các lệnh cập nhật gây ra xung đột ở cấp domain.

## Bối cảnh và vấn đề

Trong các kiến trúc truyền thống, truy vấn và cập nhật một cơ sở dữ liệu sử dụng cùng một data model. Việc này đơn giản và hoạt động tốt với các phương thức CRUD cơ bản. Nhưng trong các ứng dụng phức tạp, phương pháp này có thể trở nên khó sử dụng. Ví dụ, ở phía đọc, ứng dụng có thể sử lý nhiều truy vấn khác nhau và trả về các data transfer object (DTO) với các hình thái khác nhau. Việc ánh xạ đối tượng có thể trở nên phức tạp. Ở phía ghi, model có thể thực hiện valiidatio và logic nghiệp vụ phức tạp. Kết quả là, bạn có thể kết thúc với một model quá phức tạp và làm quá nhiều thứ.

Khối lượng công việc đọc và ghi thường không đối xứng với các yêu cầu về hiệu suất và quy mô rất khác nhau.



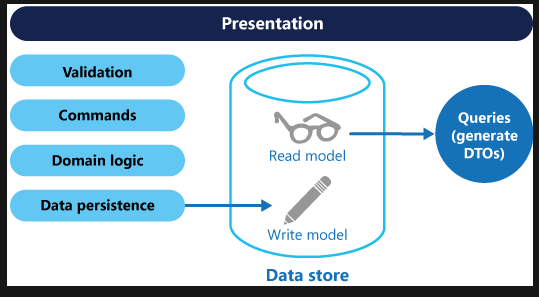
* Thường có sự không khớp giữa các sự biểu thị dữ liệu giữa đọc và ghi, ví dụ như các cột hoặc các thuộc tính bổ sung phải được cập nhật chính xác mặc dù chúng không được yêu cầu như một phần của một phương thức.
* Tranh trấp dữ liệu có thể xảy ra khi các phương thức được thực thi song song trên cùng tập hợp dữ liệu.
* Phương pháp truyền thống có thể có tác động tiêu cực đến hiệu năng do tải treenn lopwss lưu trữ dữ liệu và truy cập dữ liệu cũng như sự phức tạp của các truy vấn cần thiết để truy vấn thông tin.
* Việc quản lý bảo mật và các phân quyền có thể trở lên phức tạp vì mỗi entity phải thực hiện cả hoạt động đọc và ghi, điều này có thể hiển thị dữ liệu trong ngữ cảnh sai (wrong context).

## SGiải pháp

CQRS phân tách đọc và ghi thành các model khác nhau bằng việc dùng các command để cập nhật dữ liệu, và các query để đọc dữ liệu.

* Các command nên được dựa trên nhiệm vụ thay vì tập trung vào dữ liệu. (“đặt phòng khách sạn”, không phải “Đặt trạng thái đặt phòng thành đã đặt trước”).
* Các command có thể được đặt trên một hang chờ để sử lý bất đồng bộ, thay vì được sử lý đồng bộ.
* Các query không bao giờ được thay đổi database. Một query trả về một DTO không được bao gồm thông tin domain.

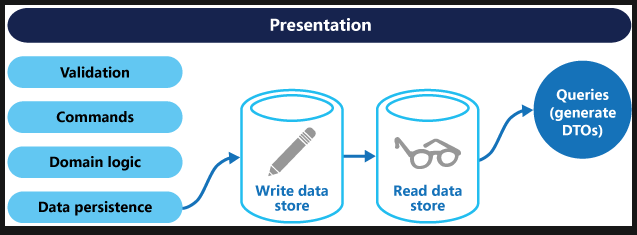
Các model sau đó có thể đươc tách biệt, nhưng hình sau, mặc dù nó không phải là một yêu cầu tuyệt đối.



Việc có các model truy vẫn và cập nhật riêng biệt sẽ đơn giản hóa việc thiết kế và triển khai. Tuy nhiên, một hạn chế là code CQRS không thể tự động đươc sinh ra từ một lược đồ cơ sở dữ liệu sử dụng các cơ chế scaffolding như O/RM tools.

Để tách biệt tốt hơn, bạn có thể phân tách dữ liệu đọc từ dữ liệu ghi. Trong trường hợp đó, cơ sở dữ liệu đọc có thể sử dụng lược đồ dữ liệu của riêng nó được tối ưu hóa cho các truy vấn. Ví dụ, nó có thể lưu trữ một dạng xem cụ thể hóa ([materiaid view](https://docs.microsoft.com/en-us/azure/architecture/patterns/materialized-view)) của dữ liệu, để tránh các join phức tạp hoặc các tham chiếu phức tạp. Nó thậm chí thể sử dụng một kiểu lưu trữ dữ liệu khác. Ví dụ, cơ sở dũ liệu ghi có thể là dạng quan hệ, trong khi cơ sở dữ liệu đọc là dạng tài liệu.

Nếu các cơ sở dữ liệu đọc và ghi riêng biệt được sử dụng, chúng phải được giữ đồng bộ. Thông thường, điều này được thực hiện thông qua việc có model ghi truyền đi một event mỗi khi nó cập nhật cơ sở dữ liệu. Để biết them thông tin về việc sử dụng các event, tham khảo thêm [Event-driven architecture style](https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven). Việc cập nhật cơ sở dữ liệu và truyền đi eent phải được xảy ra trong một transaction duy nhất.



# Messaging

# Circuit break

# Saga

# Domain event

# Event sourcing

# Log aggregate

# Audit logging

# Tracing