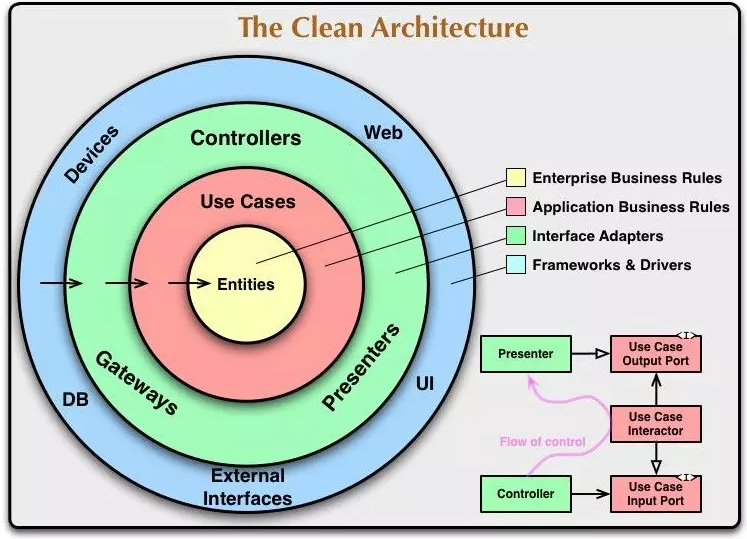
# Clean Architecture là gì?

Clean Architecture được xây dựng dựa trên tư tưởng "độc lập" kết hợp với các nguyên lý thiết kế hướng đối tượng (đại diện tiêu biểu là Dependency Inversion). Độc lập ở đây nghĩa là việc project không bị phụ thuộc vào framework và các công cụ sử dụng trong quá trình kiểm thử.

Kiến trúc của Clean Architecture chia thành 4 layer với một quy tắc phụ thuộc. Các layer bên trong không nên biết bất kỳ điều gì về các layer bên ngoài. Điều này có nghĩa là nó có quan hệ phụ thuộc nên "hướng" vào bên trong. Nhìn vào hình vẽ minh họa sau đây:



***Entities***: là khái niệm dùng để mô tả các Business Logic. Đây là layer quan trọng nhất, là nơi bạn thực hiện giải quyết các vấn đề - mục đích khi xây dựng app. Layer này không chứa bất kỳ một framework nào, bạn có thể chạy nó mà không cần emulator. Nó giúp bạn dễ dàng test, maintain và develop phần business logic.

***Use case*** : chứa các rule liên quan trực tiếp tới ứng dụng cục bộ (application-specific business rules).

***Interface Adapter*** : tập hợp các adapter phục vụ quá trình tương tác với các công nghệ.

***Framework and Drivers*** : chứa các công cụ về cơ sở dữ liệu và các framework, thông thường bạn sẽ không phải lập trình nhiều ở tầng này. Tuy nhiên cần chắc chắn về mức ưu tiên sử dụng các công cụ này trong project.

***Code của các class thuộc lớp trong không được tham chiếu đến code của class thuộc lớp ngoài.***

Thông thường thì một ứng dụng của bạn có thể có tùy ý số lượng các layer. Thường thì một ứng dụng sẽ có 3 layer:

* **Outer**: Implementation layer: là nơi mà tất cả mọi thứ của framwork xảy ra, điều này bao gồm tất cả các công cụ Android như là tạo các activity, các fragment, gửi intent, networking và databases.
* **Middle**: Interface adapter layer: là hoạt động như một kết nối giữa business logic và framework specific code.
* **Inner**: Business logic layer: tương tự như trên.

Đối với mỗi layer ở trên core layer đều có trách nhiệm chuyển đổi các model thành các model layer thấp hơn trước khi các layer thấp hơn có thể sử dụng đến chúng. Tại sao việc chuyển đổi model là cần thiết? Ví dụ, các business logic model của bạn có thể không thích hợp cho việc hiển thị chúng đối với người dùng cuối, bạn có thể sẽ phải kết hợp nhiều nhiều business logic model cùng một lúc. Vì vậy, bạn nên tạo một lớp ViewModel để có thể dễ dàng hiển thị UI. Sau đó, hãy sử dụng một lớp converter ở outer layer để chuyển đổi các business model của bạn sao cho thích hợp với ViewModel. Tóm lại là chuyển đổi Model để phù hợp với chức năng của từng layer!

**Tham khảo thêm:** [**clean architecture with asp.net core 6**](https://vi.wikipedia.org/wiki/Mã_spaghetti)

# Domain-Driven design (DDD)

Domain-driven design (DDD) ủng hộ việc lập mô hình dựa trên nghiệp vụ thực tế có liên quan đến các use case của bạn. Trong bối cảnh xây dựng các ứng dụng, DDD nói về các vấn đề dưới dạng các domain. Nó mô tả các khu vực vấn đề độc lập dưới dạng bounded context (mỗi bounded context tương quan với một microservice) và nhấn mạnh một ngôn ngữ chung để nói về những vấn đề này. Nó cũng gợi ý nhiều khái niệm và pattern kỹ thuật, như các domain entity với các rich model (không có [anemic-domain model](https://martinfowler.com/bliki/AnemicDomainModel.html)), các value object, aggregate và quy tắc aggregate root (hoặc root entity) để hỗ trợ việc triển khai nội bộ. Phần này giới thiệu thiết kế và thực hiện các pattern nội bộ đó.

Đôi khi những quy tắc và pattern kỹ thuật DDD này được coi là những trở ngại có một đường cong học tập dốc để thực hiện các phương pháp tiếp cận DDD. Nhưng phần quan trọng không phải là bản thân các pattern, mà là tổ chức code sao cho phù hợp với các vấn đề nghiệp vụ và sử dụng các thuật ngữ nghiệp vụ giống nhau. Ngoài ra, các phương pháp tiếp cận DDD chỉ nên được áp dụng nếu bạn đang triển khai các microservice phức tạp với các quy tắc nghiệp vụ quan trọng. Các trách nhiệm đơn giản hơn, như dịch vụ CRUD, có thể được quản lý bằng các cách tiếp cận đơn giản hơn.

Tìm ra các boundary là nhiệm vụ quan trọng khi thiết kế và xác định một microservice. Các DDD pattern giúp bạn hiểu được độ phức tạp trong domain. Đối với domain model cho mỗi bounded context, bạn định nghĩa và xác định các thực thể, value object và aggregate mà mô hình hóa domain của bạn. Bạn xây dựng và tinh chỉnh domain model nằm trong bounded context của bạn. Và điều đó rõ ràng dưới dạng một microservice. Các thành phần trong các boundary đó cuối cùng sẽ là microservices của bạn, mặc dù trong một số trường hợp, BC hoặc microservices nghiệp vụ có thể bao gồm một số dịch vụ vật lý. DDD là về boundary và microservices cũng vậy.

## **Giữ cho các giới hạn ngữ cảnh của microservice thật nhỏ**

Xác định vị trí đặt giới hạn giữa các bounded context cân bằng hai mục tiêu mâu thuẫn. Đầu tiên, ban đầu bạn muốn tạo các microservices nhỏ nhất có thể, mặc dù đó không phải là trình điều khiển chính; bạn nên tạo ra một giới hạn xung quanh những thứ cần sự gắn kết. Thứ hai, bạn muốn tránh giao tiếp trò chuyện giữa các microservices. Những mục tiêu này có thể mâu thuẫn với nhau. Bạn nên cân bằng chúng bằng cách phân tách hệ thống thành nhiều microservices nhỏ nhất có thể cho đến khi bạn thấy giới hạn giao tiếp phát triển nhanh chóng với mỗi nỗ lực bổ sung để tách một bounded context mới. Sự liên kết là chìa khóa trong một bounded context duy nhất.

Nó tương tự như mã không phù hợp khi triển khai các lớp. Nếu hai microservice cần làm việc nhiều với nhau, chúng có thể phải là cùng một microservice.

Một cách khác để nhìn vào khía cạnh này là tính độc lập. Nếu một microservice phải dựa vào một service khác để phục vụ trực tiếp một request, thì nó không thực sự có tính độc lập.

## **Các lớp trong DDD Microservice**

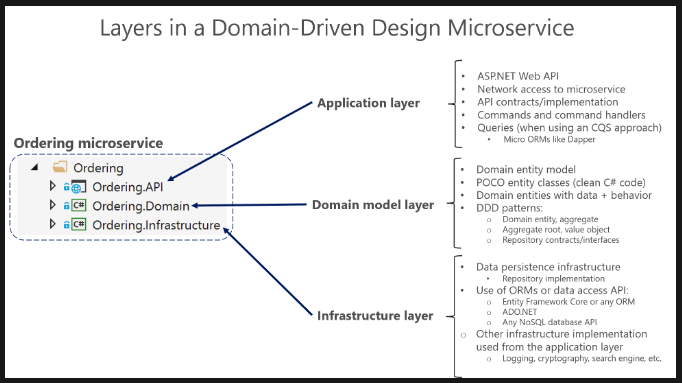
Hầu hết các ứng dụng doanh nghiệp có độ phức tạp nghiệp vụ và kỹ thuật đáng kể được xác định bởi nhiều lớp(layer). Các lớp là một thành phần logic và không liên quan đến việc triển khai dịch vụ. Chúng tồn tại để giúp các developer quản lý độ phức tạp trong code. Các lớp khác nhau (như domain model layer so với presentation layer, v.v.) có thể có các kiểu khác nhau, bắt buộc phải dịch giữa các kiểu đó.

Ví dụ, một thực thể có thể được tải từ cơ sở dữ liệu. Sau đó, một phần của thông tin đó aggregate thông tin bao gồm dữ liệu bổ sung từ các thực thể khác, có thể được gửi đến giao diện người dùng của ứng dụng khách thông qua REST Web API. Vấn đề ở đây là domain entity được chứa trong domain model layer và không được truyền sang các khu vực khác mà nó không thuộc về, chẳng hạn như presentation layer.

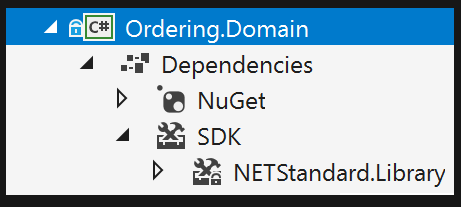
Ngoài ra, bạn cần có các thực thể luôn hợp lệ (xem phần [design validations in domain layer](https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate/Order.cs)) được kiểm soát bởi các aggregate root (các root entity). Do đó, các thực thể không nên bị ràng buộc với client, vì ở cấp giao diện người dùng, một số dữ liệu có thể vẫn chưa được xác thực. Đó là lý do chúng ta có ViewModel. ViewModel là một mô hình dữ liệu dành riêng cho nhu cầu của presentation layer. Các domain entity không trực tiếp thuộc về ViewModel. Thay vào đó, bạn cần truyển đổi giữa các ViewModels với các domain entity và ngược lại.

Khi giải quyết độ phức tạp, điều quan trọng là phải có một domain model được kiểm soát bởi các aggregate root để đảm bảo rằng tất cả các bất biến và quy tắc liên quan đến nhóm thực thể đó (aggregate) được thực hiện thông qua một điểm vào hoặc cổng duy nhất, aggregate root.

Hình sau cho thấy cách một thiết kế phân lớp được triển khai trong ứng dụng eShopOnContainers.



Ba layer trong một microservice DDD như Ordering. Mỗi layer là một VS project: Application layer là Ordering.API, Domain layer là Ordering.Domain và Infrastructure layer là Ordering.Infrastructure. Bạn muốn thiết kế hệ thống sao cho mỗi layer chỉ giao tiếp với một số layer khác. Cách tiếp cận đó có thể dễ thực thi hơn nếu các layer được triển khai dưới dạng các thư viện lớp khác nhau, bởi vì bạn có thể xác định rõ ràng những phụ thuộc nào được đặt giữa các thư viện. Ví dụ: domain model layer không được phụ thuộc vào bất kỳ layer nào khác (các lớp domain model phải là các lớp Đối tượng Lớp cũ thuần túy hoặc POCO, các lớp). Như trong Hình sau, Ordering.Domain layer chỉ phụ thuộc vào các thư viện .NET hoặc gói NuGet, chứ không phụ thuộc vào bất kỳ thư viện tùy chỉnh nào khác, chẳng hạn như thư viện dữ liệu hoặc thư viện persistence.



## **Domain model layer**

Cuốn sách [Domain Driven Design](https://www.domainlanguage.com/ddd/) của Eric Evans nói như sau về domain model layer và application layer.

Domain model layer: Chịu trách nhiệm trình bày các khái niệm về nghiệp vụ, thông tin về tình trạng nghiệp vụ và các quy tắc nghiệp vụ. Trạng thái phản ánh tình hình nghiệp vụ được kiểm soát và sử dụng ở đây, mặc dù các chi tiết kỹ thuật của việc lưu trữ nó được giao cho Infrastructure. Layer này là trung tâm của phần mềm.

Domain model layer là nơi nghiệp vụ được thể hiện. Khi bạn triển khai domain model layer microservice trong .NET, layer đó được code như một thư viện lớp với các domain entity nắm bắt dữ liệu cùng với hành vi (các phương thức có logic).

Tuân theo các nguyên tắc [Persistence Ignorance](https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/nosql-database-persistence-infrastructure) và [Infrastructure Ignorance](https://ayende.com/blog/3137/infrastructure-ignorance)**,** layer này phải hoàn toàn bỏ qua các chi tiết về persistence dữ liệu. Các tác vụ persistence này sẽ được thực hiện bởi infrastructure layer. Do đó, layer này không nên phụ thuộc trực tiếp vào infrastructure, có nghĩa là một quy tắc quan trọng là các lớp domain model entity của bạn phải là POCO.

Các domain entity không được có bất kỳ sự phụ thuộc trực tiếp nào (như dẫn xuất từ ​​một lớp cơ sở) vào bất kỳ data access infrastructure framework nào như Entity Framework hoặc NHibernate. Tốt nhất, các domain entity của bạn không nên bắt nguồn hoặc triển khai từ các kiểu được xác định trong bất kỳ infrastructure framework nào.

Hầu hết các ORM framework hiện đại như Entity Framework Core đều cho phép cách tiếp cận này, để các lớp domain model của bạn không được kết hợp với infrastructure. Tuy nhiên, việc có các thực thể POCO không phải lúc nào cũng khả thi khi sử dụng một số cơ sở dữ liệu và khuôn khổ NoSQL nhất định, như Actors và Reliable Collections trong Azure Service Fabric.

Ngay cả khi điều quan trọng là phải tuân theo nguyên tắc Persistence ignorance về tính bền bỉ đối với domain model của bạn, bạn cũng không nên bỏ qua các vấn đề về persistence. Điều quan trọng vẫn là hiểu mô hình dữ liệu vật lý và cách nó ánh xạ đến mô hình đối tượng thực thể của bạn. Nếu không, bạn có thể tạo ra những thiết kế bất khả thi.

Ngoài ra, khía cạnh này không có nghĩa là bạn có thể lấy một mô hình được thiết kế cho cơ sở dữ liệu quan hệ và chuyển trực tiếp nó sang NoSQL hoặc cơ sở dữ liệu hướng tài liệu. Trong một số mô hình thực thể, mô hình có thể phù hợp, nhưng thường thì không. Vẫn có những ràng buộc mà mô hình thực thể của bạn phải tuân thủ, dựa trên cả công nghệ lưu trữ và công nghệ ORM.

## **Application layer**

Chuyển sang application layer, chúng ta có thể trích dẫn lại cuốn sách [Domain Driven Design](https://martinfowler.com/bliki/AnemicDomainModel.html) của Eric Evans:

Application layer: Xác định các công việc mà phần mềm phải làm và hướng các đối tượng domain biểu đạt xử lý các vấn đề. Các tác vụ mà layer này chịu trách nhiệm có ý nghĩa đối với nghiệp vụ hoặc cần thiết cho sự tương tác với các application layer của các hệ thống khác. layer này được giữ mỏng. Nó không chứa các quy tắc hoặc kiến ​​thức nghiệp vụ, mà chỉ điều phối các task và delegate của các đối tượng domain ở layer tiếp theo trở xuống. Nó không có trạng thái phản ánh trạng thái nghiệp vụ, nhưng nó có thể có trạng thái phản ánh tiến độ của một task cho người dùng hoặc chương trình.

Application layer của microservice trong .NET thường được code như là một dự án ASP.NET Core Web API. Dự án triển khai tương tác của microservice, truy cập mạng từ xa và các API Web bên ngoài được sử dụng từ giao diện người dùng hoặc ứng dụng khách. Nó bao gồm các query nếu sử dụng cách tiếp cận CQRS, command được microservice chấp nhận và thậm chí cả giao tiếp theo hướng sự kiện giữa các microservices (integration event). ASP.NET Core Web API đại diện cho application layer không được chứa các quy tắc nghiệp vụ hoặc kiến ​​thức domain (đặc biệt là các quy tắc domain cho các transaction hoặc cập nhật); chúng phải được sở hữu bởi thư viện domain model layer. Application layer chỉ được điều phối các tác vụ và không được giữ hoặc xác định bất kỳ trạng thái domain nào (domain model). Nó ủy quyền việc thực thi các quy tắc nghiệp vụ cho chính các lớp domain model(aggregate root và các domain entity), cuối cùng sẽ cập nhật dữ liệu trong các domain entity đó.

Về cơ bản, logic ứng dụng là nơi bạn triển khai tất cả các use case phụ thuộc vào giao diện người dùng nhất định. Ví dụ: việc triển khai liên quan đến dịch vụ API Web.

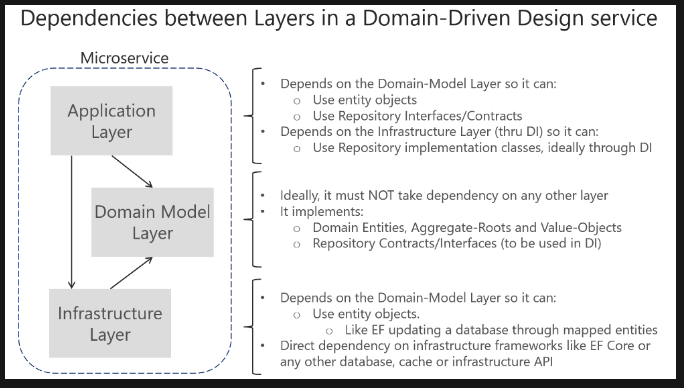
Mục tiêu là logic miền trong domain model layer, các bất biến của nó, data model và các quy tắc nghiệp vụ liên quan phải hoàn toàn độc lập với các presentation layer và application layer. Trên hết, Domain model layer không được phụ thuộc trực tiếp vào bất kỳ infrastructure framework nào.

## **Infrastructure layer**

Infrastructureayer là cách dữ liệu được lưu giữ ban đầu trong các domain entity (trong bộ nhớ) được duy trì trong cơ sở dữ liệu hoặc một kho lưu trữ liên tục khác. Một ví dụ điển hình là sử dụng code Core Entity Framework để triển khai các lớp repository pattern sử dụng DBContext để duy trì dữ liệu trong cơ sở dữ liệu quan hệ.

Phù hợp với các nguyên tắc [Persistence Ignorance](https://github.com/jbogard/MediatR) và [Infrastructure Ignorance](https://ayende.com/blog/3137/infrastructure-ignorance) đã đề cập trước đó, infrastructure layer không được "làm ô nhiễm" domain model layer. Bạn phải giữ cho các lớp domain model antity bất khả tri khỏi infrastructure mà bạn sử dụng để persist dữ liệu (EF hoặc bất kỳ framework nào khác) bằng cách không sử dụng các phụ thuộc cứng vào các framework. Thư viện domain model layer của bạn chỉ nên có code domain của bạn, chỉ các lớp thực thể POCO triển khai trung tâm của phần mềm của bạn và hoàn toàn tách biệt khỏi các công nghệ infrastructure.

Do đó, các layer hoặc thư viện lớp và các project của bạn cuối cùng sẽ phụ thuộc vào domain model layer (thư viện) của bạn, chứ không phải ngược lại, như trong Hình sau.



Sự phụ thuộc trong DDD service, application layer phụ thuộc vào Domain layer và Infrastructure layer và Infrastructure layer phụ thuộc vào Domain layer, nhưng Domain layer không phụ thuộc vào bất kỳ layer nào. Thiết kế layer này phải độc lập cho từng microservice. Như đã lưu ý trước đó, bạn có thể triển khai các microservice phức tạp nhất theo các DDD pattern, trong khi triển khai các microservice theo hướng dữ liệu đơn giản hơn (CRUD đơn giản trong một lớp) theo cách đơn giản hơn.

# Design a microservice domain model

Xác định một domain model phong phú cho mỗi nghiệp vụ microservice hoặc bounded context

Mục tiêu của phần này là giúp bạn tạo domain model cho từng nghiệp vụ microservice hoăc Bounded Context (BC). Tuy nhiên, hãy nhớ rằng BC hoặc nghiệp vụ microservice đôi khi cos thể bao gồm một số dịch vụ vật lý chia sẻ một domain model đơn. Domain model phải nắm bắt các quy tắc, hành vi, ngôn ngữ nghiệp vụ và các ràng buộc của bounded context duy nhất hoặc nghiệp vụ microservice mà nó đại diện.

## **The Domain Entity pattern**

Các thực thể đại diện cho các đối tượng miền (domain object) và chủ yếu được xác định bởi định danh (identity), tính liên tục và bền bỉ của chúng theo thời gian, chứ không chỉ bởi các thuộc tính bao gồm chúng. Như Eric Evans nói, "một đối tượng chủ yếu được xác định bởi định danh của nó được gọi là Thực thể." Các thực thể rất quan trọng trong domain model, vì chúng là cơ sở cho một model. Vì vậy, bạn nên xác định và thiết kế chúng một cách cẩn thận

Định danh của một thực thể có thể trải dài qua nhiều microservices hoặc Bound Contexts.

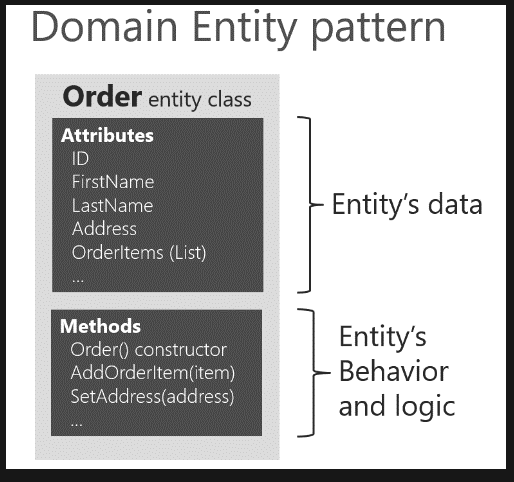
Cùng một định danh (nghĩa là cùng một giá trị Id, mặc dù có thể không phải là cùng một domain entity) có thể được model hóa trên nhiều bounded context hoặc microservices. Tuy nhiên, điều đó không có nghĩa là cùng một thực thể, với các thuộc tính và logic giống nhau sẽ được triển khai trong nhiều bounded context. Thay vào đó, các thực thể trong mỗi bounded context giới hạn các thuộc tính và hành vi của chúng đối với những gì được yêu cầu trong domain của bounded context đó

Ví dụ: thực thể Buyer có thể có hầu hết các thuộc tính của một người được xác định trong thực thể User trong profile hoặc identity microservice, bao gồm cả định danh. Tuy nhiên, Buyer trong microservice đặt hàng có thể có ít thuộc tính hơn, vì chỉ một số dữ liệu người mua nhất định có liên quan đến quy trình đặt hàng. Bối cảnh của mỗi microservice hoặc bounded context tác động đến domain model của nó.

Domain entity phải thực hiện hành vi ngoài việc triển khai các thuộc tính dữ liệu.

Một domain entity trong DDD phải triển khai domain logic hoặc hành vi liên quan đến dữ liệu thực thể (đối tượng được truy cập trong bộ nhớ). Ví dụ: là một phần của order entity class, bạn phải triển khai logic nghiệp vụ và hoạt động dưới dạng các phương thức cho các tác vụ như thêm order item, validate dữ liệu và tính tổng. Các phương thức của thực thể quan tâm đến các bất biến và các quy tắc của thực thể thay vì để các quy tắc đó trải rộng trên application layer.

Hình sau cho thấy một domain entity thực hiện không chỉ các thuộc tính dữ liệu mà còn các phép toán hoặc phương thức với domain logic liên quan.



Một domain entity model thực hiện các hành vi thông qua các phương thức, nghĩa là, nó không phải là một "anemic" model. Tất nhiên, đôi khi bạn có thể có các thực thể không triển khai bất kỳ logic nào như một phần của lớp thực thể. Điều này có thể xảy ra trong các thực thể con trong một aggregate nếu thực thể con không có bất kỳ logic đặc biệt nào vì hầu hết logic được xác định trong aggregate root. Nếu bạn có một microservice phức tạp có logic được triển khai trong các lớp dịch vụ thay vì trong các domain entity, bạn có thể rơi vào "anemic" model, được giải thích trong phần sau.

## **Rich domain model so với anemic domain model**

Trong bài đăng [AnemicDomainModel](https://deviq.com/principles/persistence-ignorance) của mình, Martin Fowler mô tả "anemic" domain model theo cách này:

Biểu hiện cơ bản của "anemic" domain model là lúc đầu, nó trông giống như domain model thật. Có những đối tượng, nhiều đối tượng được đặt tên theo danh từ trong không gian domain, và những đối tượng này được kết nối với các mối quan hệ và cấu trúc phong phú mà các domain model thực sự có. Có thể nhận ra được khi bạn nhìn vào hành vi và bạn nhận ra rằng hầu như không có bất kỳ hành vi nào trên những đối tượng này, khiến chúng chỉ hơn một gói setter và getter.

Tất nhiên, khi bạn sử dụng domain model khác, các data model đó sẽ được sử dụng từ một tập hợp các service object (được đặt tên theo truyền thống là lớp nghiệp vụ) nắm bắt tất cả domain hoặc logic nghiệp vụ. Lớp nghiệp vụ nằm trên đầu data model và sử dụng mô hình dữ liệu giống như dữ liệu.

Anemic domain model chỉ là một thiết kế theo phong cách thủ tục. Các đối tượng anemic entity không phải là đối tượng thực vì chúng thiếu hành vi (phương thức). Chúng chỉ giữ các thuộc tính dữ liệu và do đó nó không phải là thiết kế hướng đối tượng. Bằng cách đưa tất cả các hành vi vào các service object (lớp nghiệp vụ), về cơ bản, bạn sẽ kết thúc với [mã spaghetti](https://www.dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf) hoặc các [transactionScript](https://martinfowler.com/eaaCatalog/transactionScript.html), và do đó bạn mất đi những lợi thế mà domain model cung cấp.

Bất kể, nếu microservice hoặc bounded context của bạn rất đơn giản (dịch vụ CRUD), thì anemic domain model ở dạng các đối tượng thực thể chỉ có thuộc tính dữ liệu là tốt hơn so với việc triển khai các mẫu DDD phức tạp hơn. Trong trường hợp đó, nó sẽ chỉ đơn giản là một mô hình bền vững, bởi vì bạn đã cố ý tạo một thực thể chỉ có dữ liệu cho mục đích CRUD.

Đó là lý do tại sao các kiến ​​trúc microservices hoàn hảo cho cách tiếp cận đa kiến ​​trúc tùy thuộc vào từng bounded context. Ví dụ: trong eShopOnContainers, ordering microservice triển khai các mẫu DDD, nhưng catalog microservice, là một dịch vụ CRUD đơn giản, thì không.

Một số người nói rằng anemic domain model là một phản mô hình (anti-pattern). Nó thực sự phụ thuộc vào những gì bạn đang thực hiện. Nếu microservice bạn đang tạo đủ đơn giản (ví dụ: dịch vụ CRUD), thì theo mô hình miền thiếu máu, nó không phải là mô hình chống. Tuy nhiên, nếu bạn cần giải quyết sự phức tạp của miền của microservice có nhiều quy tắc kinh doanh luôn thay đổi, thì mô hình miền thiếu máu có thể là một mô hình chống lại microservice đó hoặc Bối cảnh bị ràng buộc. Trong trường hợp đó, thiết kế nó như một mô hình phong phú với các thực thể chứa dữ liệu cộng với hành vi cũng như triển khai các mẫu DDD bổ sung (tổng hợp, đối tượng giá trị, v.v.) có thể mang lại lợi ích to lớn cho sự thành công lâu dài của một dịch vụ nhỏ như vậy.

## **Value Object pattern**

Như Eric Evans đã lưu ý, "Nhiều đối tượng không có định danh khái niệm. Những đối tượng này mô tả các đặc điểm nhất định của một sự vật."

Một thực thể yêu cầu một định danh, nhưng có nhiều đối tượng trong một hệ thống thì không, như mẫu value object. Value object là một đối tượng không có định danhkhais niệm mà mô tả một khía cạnh domain. Đây là những đối tượng mà bạn khởi tạo để đại diện cho các phần tử thiết kế mà bạn chỉ quan tâm tạm thời. Bạn quan tâm đến chúng là gì chứ không phải là cái gì. Ví dụ bao gồm số và chuỗi, nhưng cũng có thể là các khái niệm cấp cao hơn như nhóm thuộc tính.

Một thực thể trong một microservice này có thể không phải là một thực thể trong một microservice khác, bởi vì trong trường hợp thứ hai, bounded context có thể có một ý nghĩa khác. Ví dụ: một address trong ứng dụng thương mại điện tử có thể hoàn toàn không có danh tính, vì nó có thể chỉ đại diện cho một nhóm thuộc tính trong hồ sơ của khách hàng cho một người hoặc công ty. Trong trường hợp này, Address nên được phân loại là một value object. Tuy nhiên, trong đơn đăng ký cho một công ty điện lực, Address khách hàng có thể quan trọng đối với lĩnh vực kinh doanh. Do đó, Address chỉ phải có định danh để hệ thống thanh toán có thể được liên kết trực tiếp với Address. Trong trường hợp đó, một Address nên được phân loại là một domain entity.

Một người có tên và họ thường là một thực thể vì một người có định danh, ngay cả khi tên và họ trùng với một bộ giá trị khác, chẳng hạn như nếu những tên đó cũng đề cập đến một người khác.

Các đối tượng giá trị khó quản lý trong cơ sở dữ liệu quan hệ và ORM như Entity Framework (EF), trong khi trong cơ sở dữ liệu hướng tài liệu, chúng dễ triển khai và sử dụng hơn.

Ở EF Core 2.0 và các phiên bản mới hơn bao gồm tính năng [owned entity](https://udidahan.com/2008/08/25/domain-events-take-2/#owned-entities-and-table-splitting) giúp xử lý các value object dễ dàng hơn.

## **The Aggregate pattern**

Domain model chứa các cụm thực thể dữ liệu và sử lý khác nhau mà có thể kiểm soát một mảng chức năng quan trọng, chẳng hạn như ordering hoặc inventory. Một đơn vị DDD chi tiết hơn là aggregate, mô tả một cụm hoặc nhóm các thực thể và hành vi có thể được coi như một đơn vị gắn kết.

Bạn thường xác định aggregate dựa trên các transaction mà bạn cần. Một ví dụ điển hình là một order cũng chứa danh sách các order item. Một order item thường sẽ là một thực thể. Nhưng nó sẽ là một thực thể con trong order aggregate, cũng sẽ chứa thực thể order làm root entity của nó, thường được gọi là aggregate root.

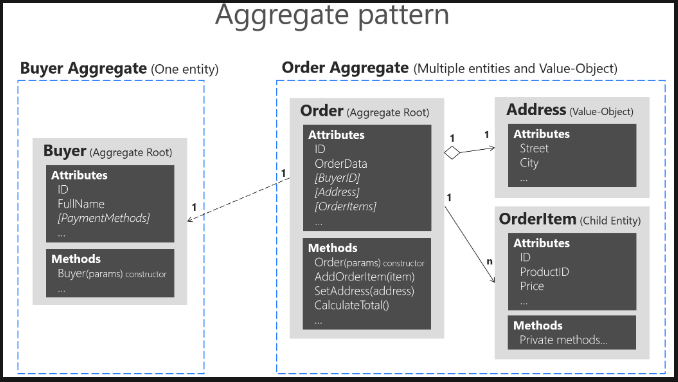
Việc xác định các aggregate có thể khá khó khăn. Aggregate là một nhóm các đối tượng phải nhất quán với nhau, nhưng bạn không thể chỉ chọn một nhóm đối tượng và gắn nhãn chúng là một aggregate. Bạn phải bắt đầu với khái niệm domain và nghĩ về các thực thể được sử dụng trong các transaction phổ biến nhất liên quan đến khái niệm đó. Những thực thể cần phải nhất quán về transaction là những gì tạo thành một aggregate. Suy nghĩ về các hoạt động transaction có lẽ là cách tốt nhất để xác định các aggregate.

## **The Aggregate Root hay Root Entity pattern**

Một aggregate bao gồm ít nhất một thực thể: aggregate root, còn được gọi là root entity hoặc thực thể chính. Ngoài ra, nó có thể có nhiều thực thể con và các value object, với tất cả các thực thể và đối tượng làm việc cùng nhau để thực hiện các hành vi và transaction được yêu cầu.

Mục đích của một aggregate root là để đảm bảo tính nhất quán của aggregate; nó phải là điểm vào duy nhất để cập nhật cho aggregate thông qua các method hoặc action trong lớp aggregate root. Bạn chỉ nên thực hiện các thay đổi đối với các thực thể trong aggregate thông qua aggregate root. Nó là người bảo vệ tính nhất quán của aggregate, xem xét tất cả các quy tắc bất biến và nhất quán mà bạn có thể cần tuân thủ trong aggregate của mình. Nếu bạn thay đổi một thực thể con hoặc value object một cách độc lập, thì aggregate root không thể đảm bảo rằng aggregate đó ở trạng thái hợp lệ. Nó sẽ giống như một cái bàn với một cái chân lỏng lẻo. Duy trì tính nhất quán là mục đích chính của aggregate root.

Trong hình sau, bạn có thể thấy các aggregate như buyer aggregate, chứa một thực thể duy nhất (buyer aggregate root). Order aggregate chứa nhiều thực thể và một đối tượng giá trị.



Một DDD domain model được tạo từ các aggregate, một aggregate có thể chỉ có một thực thể hoặc nhiều hơn và cũng có thể bao gồm các value object. Lưu ý rằng buyer aggregate có thể có thêm các thực thể con, tùy thuộc vào domain của bạn, giống như trong ordering microservice trong eShopOnContainers. Hình trên chỉ minh họa một trường hợp trong đó buyer có một thực thể duy nhất, như một ví dụ về một aggregate chỉ chứa một aggregate root.

Để duy trì sự tách biệt giữa các aggregate và giữ ranh giới rõ ràng giữa chúng, một phương pháp hay trong DDD domain model là không cho phép điều hướng trực tiếp giữa các aggregate và chỉ có trường khóa ngoại (FK), như được triển khai trong ordering microservice domain model trong [eShopOnContainers](https://github.com/dotnet-architecture/eShopOnContainers/blob/main/src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate/Order.cs). Thực thể order chỉ có trường khóa ngoại cho buyer, nhưng không có thuộc tính điều hướng EF Core, như được hiển thị trong đoạn code sau:

public class Order : Entity, IAggregateRoot

{

    private DateTime \_orderDate;

    public Address Address { get; private set; }

    private int? \_buyerId; // FK pointing to a different aggregate root

    public OrderStatus OrderStatus { get; private set; }

    private readonly List<OrderItem> \_orderItems;

    public IReadOnlyCollection<OrderItem> OrderItems => \_orderItems;

    // ... Additional code

}

Việc xác định và làm việc với các aggregate đòi hỏi phải có nghiên cứu và kinh nghiệm.

# Domain events

Sử dụng các domain event để triển khai rõ ràng các side-effect của các thay đổi trong domain của bạn. Nói cách khác, theo DDD, hãy sử dụng các domain event để triển khai rõ ràng các side-effect trên nhiều aggregate. Để có khả năng mở rộng tốt hơn và ít ảnh hưởng hơn trong việc khóa cơ sở dữ liệu, hãy sử dụng tính nhất quán cuối cùng giữa các aggregate trong cùng một domain.

## **What is a domain event?**

Một **EVENT** là một cái gì đó đã xảy ra trong quá khứ. **DOMAIN EVENT** là, một cái gì đó đã xảy ra trong Domain mà bạn muốn các phần khác của cùng một Domain (đang trong quá trình xử lý) biết. Các phần được thông báo thường phản ứng bằng cách nào đó với các **EVENT**.

Một lợi ích quan trọng của các Domain event là các side-effect có thể được thể hiện một cách rõ ràng.

Ví dụ: nếu bạn chỉ đang sử dụng Entity Framework và phải có tương tác với một số event, bạn có thể sẽ code bất kỳ thứ gì bạn cần để kích hoạt event. Vì vậy, quy tắc được kết hợp ngầm với code và bạn phải xem xét mã với hy vọng rằng quy tắc được thực hiện ở đó.

Mặt khác, sử dụng các Domain event làm cho khái niệm rõ ràng, bởi vì có một DomainEvent và ít nhất một DomainEventHandler liên quan.

Ví dụ: trong ứng dụng eShopOnContainers, khi một đơn đặt hàng được tạo, người dùng sẽ trở thành người mua, do đó, một OrderStartedDomainEvent được tạo ra và xử lý trong ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler.

Nói tóm lại, Domain event giúp bạn diễn đạt rõ ràng các quy tắc domain, dựa trên ngôn ngữ phổ biến do các chuyên gia domain cung cấp. Domain event cũng cho phép phân tách mối quan tâm tốt hơn giữa các lớp trong cùng một domain.

Điều quan trọng là phải đảm bảo rằng, giống như một transaction cơ sở dữ liệu, tất cả các hoạt động liên quan đến Domain event đều kết thúc thành công hoặc không có hoạt động nào trong số chúng thực hiện.

Các event trong domain tương tự như các event kiểu messaging, với một điểm khác biệt quan trọng. Với messaging, messaging queue, message broker hoặc service bus sử dụng AMQP, một message luôn được gửi bất đồng bộ và được truyền thông qua các quy trình và máy. Điều này rất hữu ích để tích hợp nhiều bounded context, microservices hoặc thậm chí các ứng dụng khác nhau. Tuy nhiên, với các domain event, bạn muốn tăng một event từ hoạt động domain mà bạn hiện đang chạy, nhưng bạn cũng muốn bất kỳ side-effect nào xảy ra trong cùng domain đó.

Các domain event và side-effect của chúng (các hành động được kích hoạt sau đó được quản lý bởi trình xử lý sự kiện) sẽ xảy ra gần như ngay lập tức, thường là trong quá trình và trong cùng một domain. Do đó, các domain có thể là đồng bộ hoặc không đồng bộ. Tuy nhiên, các integration event phải luôn không đồng bộ.

## **Domain events và Integration events**

Về mặt ngữ nghĩa, domain event và integration event giống nhau: thông báo về điều gì đó vừa xảy ra. Tuy nhiên, cách thực hiện của chúng phải khác. Các domain event chỉ là các thông báo được đẩy đến một bộ điều phối domain event, có thể được triển khai như một in-memory mediator dựa trên IoC container hoặc bất kỳ phương thức nào khác.

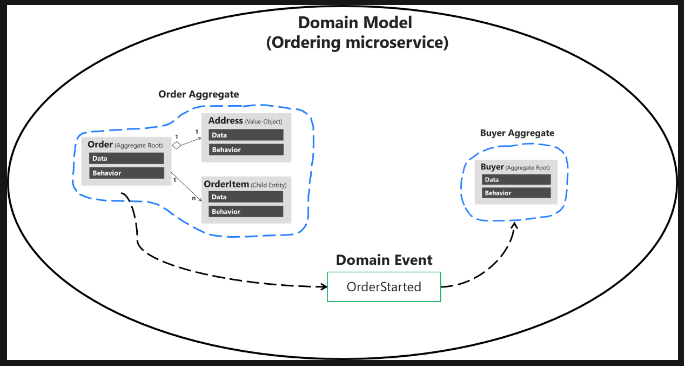
Mặt khác, mục đích của các integration event là truyền ra các transaction đã commit và các cập nhật cho các hệ thống con bổ sung, cho dù chúng là các microservice, dounded context hay thậm chí là các ứng dụng bên ngoài. Do đó, chúng sẽ chỉ xảy ra nếu entity được duy trì thành công, nếu không thì như thể toàn bộ hoạt động chưa bao giờ xảy ra.

Như đã đề cập trước đây, các integration event phải dựa trên giao tiếp không đồng bộ giữa nhiều microservices (các bounded context khác) hoặc thậm chí các hệ thống / ứng dụng bên ngoài.

Do đó, event bus interface cần một số infrastructure cho phép giao tiếp giữa các trình sử lý liên tục và phân tán giữa các dịch vụ từ xa. Nó có thể dựa trên một commercial service bus, các queues, một cơ sở dữ liệu dùng chung được sử dụng làm mailbox hoặc bất kỳ hệ thống messaging phân tán và lý tưởng nào khác.

## **Domain event như một cách hoàn hảo để kích hoạt các side-effect trên nhiều aggregate trong cùng domain**

Nếu việc thực thi một lệnh liên quan đến một instance aggregate yêu cầu các quy tắc domain bổ sung được chạy trên một hoặc nhiều instance aggregate bổ sung, bạn nên thiết kế và triển khai các side-effect đó để các domain event kích hoạt. Hình sau là một trong những trường hợp sử dụng quan trọng nhất, domain event nên được sử dụng để truyền tải các thay đổi trạng thái trên nhiều aggregate trong cùng domain model.



Hình trên cho thấy cách các domain event đạt được sự nhất quán giữa các aggregate. Khi người dùng bắt đầu một đơn đặt hàng, order aggregate sẽ gửi một domain event OrderStarted. Domain event OrderStarted do buyer aggregate xử lý để tạo đối tượng Người mua trong ordering microservice, dựa trên thông tin client từ identity microservice (với thông tin được cung cấp trong lệnh CreateOrder).

Ngoài ra, bạn có thể đăng ký aggregate root nhận các event do các thành phần của aggregate (các child entity) tạo ra. Ví dụ: mỗi thực thể con OrderItem có thể tạo event khi giá mặt hàng cao hơn một số tiền cụ thể hoặc khi số lượng mặt hàng sản phẩm quá cao. Sau đó, aggregate root có thể nhận các event đó và thực hiện phép tính hoặc tổng hợp toàn cục.

Điều quan trọng là phải hiểu rằng giao tiếp dựa trên event này không được thực hiện trực tiếp trong các aggregate; bạn cần triển khai các domain event handler.

Xử lý các domain event là một vấn đề của ứng dụng. Domain model layer chỉ nên tập trung vào logic domain — những thứ mà chuyên gia domain sẽ hiểu, chứ không phải infrastructure ứng dụng như các handler và các hành động lưu trữ dữ liệu bằng việc sử dụng các repository. Do đó, application layer level là nơi bạn nên có các domain event handler kích hoạt các hành động khi một domain event được tạo.

Domain event cũng có thể được sử dụng để kích hoạt bất kỳ số lượng hành động ứng dụng nào và điều quan trọng hơn là phải mở để tăng số lượng đó trong tương lai theo cách tách rời. Ví dụ: khi đơn đặt hàng được bắt đầu, bạn có thể muốn publish một domain event để truyền tải thông tin đó đến các aggregate khác hoặc thậm chí để tạo các hành động của ứng dụng như thông báo.

Điểm mấu chốt là số lượng mở các hành động sẽ được thực hiện khi một domain event xảy ra. Cuối cùng, các hành động và quy tắc trong domain và ứng dụng sẽ tăng lên. Độ phức tạp hoặc số lượng các hành động khi điều gì đó xảy ra sẽ tăng lên, nhưng nếu code của bạn được kết hợp với "glue" (nghĩa là tạo các đối tượng cụ thể với từ khóa new), thì mỗi khi bạn cần thêm một hành động mới, thì bạn cũng cần thay đổi code làm việc và thử nghiệm.

Thay đổi này có thể dẫn đến các lỗi mới và cách tiếp cận này cũng đi ngược lại nguyên tắc [Open / Close](https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle) của [SOLID](https://www.domainlanguage.com/ddd/). Không chỉ vậy, lớp ban đầu điều phối các hoạt động sẽ lớn hơn, điều này đi ngược lại với [Single Responsibility Principle (SRP)](https://docs.microsoft.com/en-us/events/dotnetconf-2021/clean-architecture-with-aspnet-core-6).

Mặt khác, nếu bạn sử dụng domain event, bạn có thể tạo triển khai chi tiết và tách rời bằng cách phân tách các nhiệm vụ bằng cách sử dụng phương pháp này:

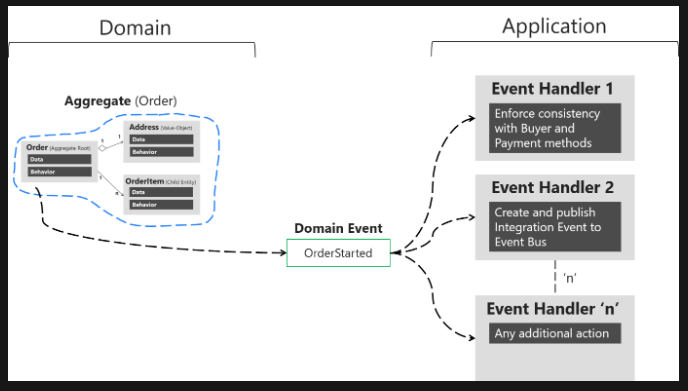
1. Gửi một lệnh (ví dụ: CreateOrder).
2. Nhận lệnh trong một command handler.

* Thực hiện giao dịch của một aggregate duy nhất.
* (Tùy chọn) tạo kiện miền cho các side-effect (ví dụ: OrderStartedDomainEvent).

1. Xử lý các domain event (trong quy trình hiện tại) sẽ thực hiện một số lượng mở các side-effect trong nhiều aggregate hoặc hành động ứng dụng. Ví dụ:

* Xác minh hoặc tạo người mua và phương thức thanh toán.
* Tạo và gửi một integration event có liên quan đến event bus để thông báo trạng thái trên các microservices hoặc kích hoạt các hành động bên ngoài như gửi email cho người mua.
* Xử lý các side-effect khác.

Như trong Hình sau, bắt đầu từ cùng một domain event, bạn có thể xử lý nhiều hành động liên quan đến các aggregate khác trong domain hoặc các hành động ứng dụng bổ sung mà bạn cần thực hiện trên các microservice kết nối với các integration event và event bus.



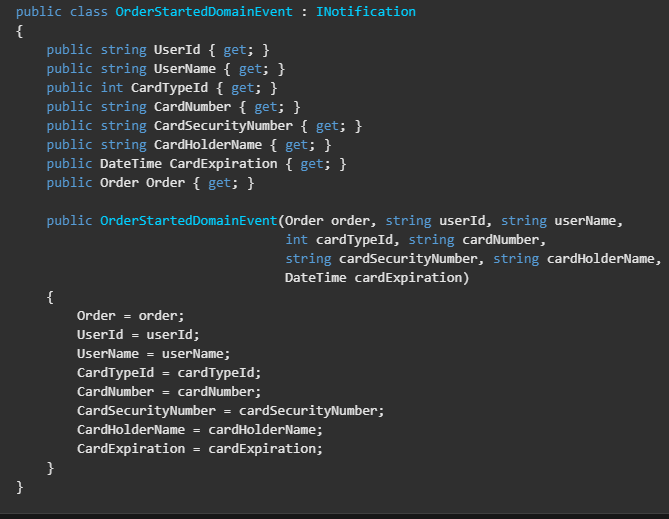
Có thể có một số hanndler cho cùng một domain event trong application layer, một handler có thể giải quyết tính nhất quán giữa các aggregate và một handler khác có thể publish một integration event, do đó các microservices khác có thể làm gì đó với nó. Các event handler thường nằm trong application layer, vì bạn sẽ sử dụng các đối tượng infrastructure như repository hoặc application API cho hành vi của microservice. Theo nghĩa đó, event handler tương tự như command handler, vì vậy cả hai đều là một phần của application layer. Sự khác biệt quan trọng là một lệnh chỉ nên được xử lý một lần. Một sự kiện miền có thể được xử lý không hoặc n lần, bởi vì nó có thể được nhận bởi nhiều receiver hoặc event handler với mục đích khác nhau với mỗi handler.

Việc có một số lượng mở các handler cho mỗi domain event cho phép bạn thêm nhiều quy tắc domain nếu cần mà không ảnh hưởng đến code hiện tại. Ví dụ: việc triển khai quy tắc nghiệp vụ sau có thể dễ dàng như thêm một vài event handler (hoặc thậm chí chỉ một):

*“Khi tổng số tiền mà khách hàng đã mua trong cửa hàng, trên bất kỳ số lượng đơn đặt hàng nào, vượt quá 6.000 đô la, hãy áp dụng chiết khấu giảm giá 10% cho mỗi đơn đặt hàng mới và thông báo cho khách hàng bằng email về mức chiết khấu đó cho các đơn đặt hàng trong tương lai.”*

## **Thực thi domain event**

Trong C #, domain event chỉ đơn giản là một cấu trúc hoặc lớp lưu giữ dữ liệu, như DTO, với tất cả thông tin liên quan đến những gì vừa xảy ra trong domain, như được minh họa trong ví dụ sau:



Đây thực chất là một lớp chứa tất cả dữ liệu liên quan đến event OrderStarted.

Về ngôn ngữ phổ biến của domain, vì một event là điều gì đó đã xảy ra trong quá khứ, tên lớp của event phải được biểu diễn dưới dạng động từ thì quá khứ, như OrderStartedDomainEvent hoặc OrderShippedDomainEvent. Đó là cách domain event được triển khai trong ordering microservice trong eShopOnContainers.

Như đã lưu ý trước đó, một đặc điểm quan trọng của các event là vì một event là một cái gì đó đã xảy ra trong quá khứ, nó sẽ không thay đổi. Do đó, nó phải là một lớp bất biến. Bạn có thể thấy trong đoạn mã trước rằng các thuộc tính ở chế độ chỉ đọc. Không có cách nào để cập nhật đối tượng, bạn chỉ có thể đặt các giá trị khi bạn tạo nó.

Điều quan trọng cần nhấn mạnh ở đây là nếu các domain event được xử lý không đồng bộ, sử dụng hàng đợi yêu cầu serialize và deserialize các đối tượng sự kiện, thì các thuộc tính sẽ phải là "private set" thay vì chỉ đọc, vì vậy deserializer sẽ có thể chỉ định các giá trị khi deserialize. Đây không phải là vấn đề trong ordering microserice, vì domain event pub / sub được triển khai đồng bộ bằng cách sử dụng MediatR.

## **Tạo domain event**

Câu hỏi tiếp theo là làm thế nào để tạo một domain event để nó tiếp cận các event handler liên quan của nó. Bạn có thể sử dụng nhiều cách tiếp cận.

Udi Dahan ban đầu đã đề xuất (ví dụ, trong một số bài đăng liên quan, chẳng hạn như [Domain event - Take 2](https://en.wikipedia.org/wiki/Single-responsibility_principle)) bằng cách sử dụng một lớp tĩnh để quản lý và tạo các event. Điều này có thể bao gồm một lớp tĩnh có tên DomainEvents sẽ tạo các domain event ngay lập tức khi nó được gọi, sử dụng cú pháp như DomainEvents.Raise (Event myEvent). Jimmy Bogard đã viết một bài đăng trên blog ([Strengthening your domain: Domain Events](https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/) )đề xuất một cách tiếp cận tương tự.

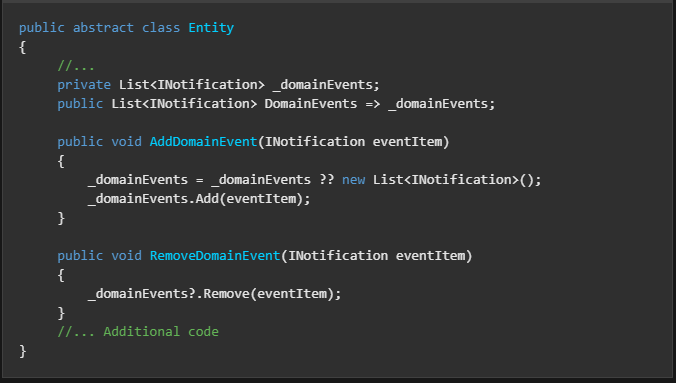
Tuy nhiên, khi lớp domain event là tĩnh, nó cũng gửi đến handler ngay lập tức. Điều này làm cho việc kiểm tra và gỡ lỗi trở nên khó khăn hơn, vì các event hander với logic các task bổ sung được thực thi ngay sau khi event được đưa ra. Khi bạn đang test và debug, bạn chỉ muốn tập trung vào những gì đang xảy ra trong các lớp aggregate hiện tại; bạn không muốn đột nhiên được chuyển hướng đến các event handler khác vì các side-effect liên quan đến các aggregate khác hoặc logic ứng dụng. Đây là lý do tại sao các phương pháp tiếp cận khác đã phát triển, như được giải thích trong phần tiếp theo.

### **Phương pháp trì hoãn để thả và điều phối các event**

Thay vì gửi đến domain event handler ngay lập tức, một phương pháp tốt hơn là thêm các domain event vào một tập hợp (collection) và sau đó gửi các domain event đó trước hoặc sau khi thực hiện transaction (như với SaveChanges trong EF). (Phương pháp này đã được Jimmy Bogard mô tả trong bài đăng này [a better domain events pattern](https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-model-layer-validations).)

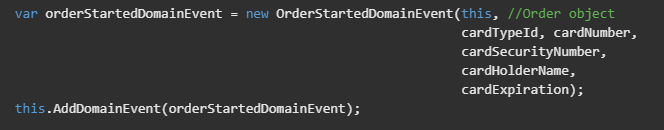
Việc quyết định xem bạn có gửi các domain event trước hay sau khi thực hiện transaction hay không là rất quan trọng, vì nó xác định xem bạn sẽ bao gồm các side-effect như một phần của cùng một transaction hay trong các transaction khác nhau. Trong trường hợp thứ hai, bạn cần phải xử lý tính nhất quán cuối cùng trên nhiều aggregate. Chủ đề này sẽ được thảo luận trong phần tiếp theo.

eShopOnContainers sử dụng phương pháp này. Trước tiên, bạn thêm các event xảy ra trong thực thể của mình vào một tập hợp hoặc danh sách các event cho mỗi thực thể. Danh sách đó phải là một phần của đối tượng thực thể, hoặc thậm chí tốt hơn, là một phần của lớp base entity của bạn, như được hiển thị trong ví dụ sau về lớp base entity:



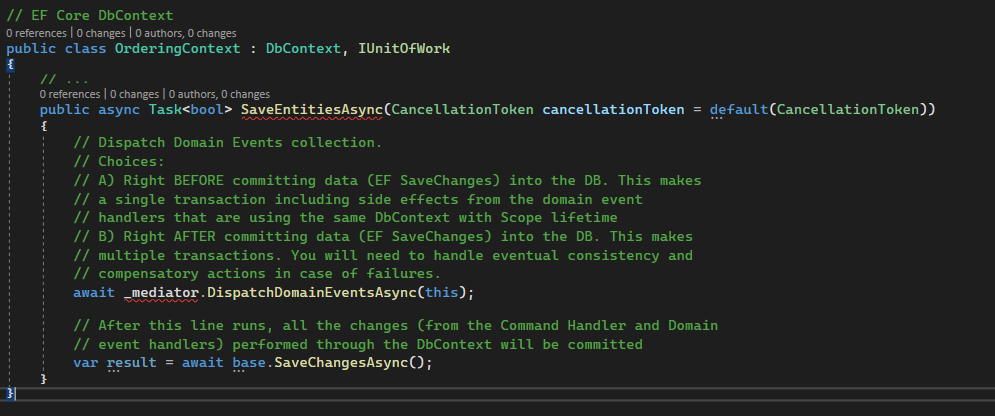
Khi bạn tạo 1 event, bạn chỉ việc thêm nó vào tập hợp event từ code đến bất kỳ phương thức nào của thực thể aggregate root.

Đoạn code sau,1 phần của [order aggregate root](https://devblogs.microsoft.com/dotnet/announcing-entity-framework-core-2-0/) trong eShopOnContainers, là một vì dụ:



Lưu ý rằng điều duy nhất mà phương thức AddDomainEvent đang làm là việc them các event vào danh sách. Không có event đã được điều phối đi, và không có event handler nào được gọi lúc này.

Bạn thực sự muốn điều phối các event sau này, khi mà bạn commit transaction với database. Nếu bạn đang sử dụng Entity Framework Core, điều đó nghĩa là phương thức SaveChanges trong EF DbContext của bạn, như đoạn code sau:



Với đoạn code này, bạn điều phối các event thực thể tới các event handler tương ứng.

Kết quả tổng thể là bạn đã tách riêng việc tạo 1 domain event (thêm vào 1 danh sách trong memory) khỏi việc điều phối nó tới 1 event handler. Ngoài ra, tùy thuộc vào kiểu điều phối bạn sử dụng, bạn có thể gửi các event bất đồng bộ hoặc đồng bộ.

Lưu ý rằng các transaction boundary đóng vai trò quan trọng ở đây. Nếu Unit of work và transaction của bạn có thể mở rộng nhiều hơn một aggregate (như khi sử dụng EF Core và cơ sở dữ liệu quan hệ), điều này có thể hoạt động tốt. Nhưng nếu transaction không thể mở rộng aggregate, bạn phải thực hiện các bước bổ sung để đạt được tính nhất quán. Đây là một lý do khác tại sao persistence ignorance không phổ biến; nó phụ thuộc vào hệ thống lưu trữ bạn sử dụng.

### **Transaction duy nhất trên các aggregate với tính nhất quán cuối cùng trên các aggregate**

Câu hỏi về việc có nên thực hiện một transaction đơn lẻ trên các aggregate hay không dựa vào sự nhất quán cuối cùng trên các aggregate đó là một câu hỏi gây tranh cãi. Nhiều tác giả DDD như Eric Evans và Vaughn Vernon ủng hộ quy tắc rằng một transaction = một aggregate và do đó lập luận về sự nhất quán cuối cùng trên các aggregate. Ví dụ, trong cuốn sách Domain-Driven Design của mình, Eric Evans nói thế này:

*“Bất kỳ quy tắc nào áp dụng cho Aggregates sẽ không được dự kiến luôn được cập nhật. Thông qua xử lý event, xử lý hàng loạt hoặc các cơ chế cập nhật khác, các vấn đề phụ thuộc khác có thể được giải quyết trong một thời gian cụ thể. (trang 128)”*

Vaughn Vernon nói như sau trong Effective [Aggregate Design. Part II: Making Aggregates Work Together](https://deviq.com/principles/persistence-ignorance)

*“Do đó, nếu việc thực thi 1 lệnh trên 1 aggregate instance yêu cầu các quy tắc nghiệp vụ bổ sung phải thực thi trên 1 hoặc nhiều aggregate, hãy sử dụng tính nhất quán cuối cùng […] Có 1 cách thuccwj tế để hỗ trợ trính nhất quán cuối cùng trong DDD. 1 phương thức aggregate publish 1 domain event đúng lúc 1 hoặc nhiều cho nhiều subscriber nhận bất đồng bộ”*

Cơ sở lý luận này dựa trên việc nắm lấy các transaction chi tiết thay vì các transaction bao gồm nhiều aggregate hoặc thực thể. Ý tưởng ở đây là trong trường hợp thứ hai, số lượng khóa cơ sở dữ liệu sẽ rất nhiều trong các ứng dụng quy mô lớn với nhu cầu khả năng mở rộng cao. Việc chấp nhận thực tế rằng các ứng dụng có khả năng mở rộng cao không cần phải có tính nhất quán transaction tức thì giữa nhiều aggregate giúp chấp nhận khái niệm về tính nhất quán cuối cùng. Các thay đổi nguyên tử thường không được cần trong nghiệp vụ, và trong mọi trường hợp các chuyên gia domain có trách nhiệm cho biết liệu các hoạt động cụ thể có cần transaction nguyên tử hay không. Nếu một hoạt động luôn cần một transaction nguyên tử giữa nhiều aggregate, bạn có thể hỏi liệu aggregate của bạn có nên lớn hơn hay nó đã không được thiết kế chính xác.

Tuy nhiên, các developer và architect khác như Jimmy Bogard không vấn đề gì với việc mở rộng một transaction duy nhất trên nhiều aggregate — nhưng chỉ khi các aggregate bổ sung đó có liên quan đến các side-effect cho cùng một lệnh ban đầu. Ví dụ: trong [A better domain events pattern](https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/), Bogard nói thế này:

*“Thông thường, tôi muốn các side-effect của domain event xảy ra trong cùng một aggregate logic, nhưng không nhất thiết phải trong cùng phạm vi tạo domain event [...] Ngay trước khi chúng tôi thực hiện transaction của mình, chúng tôi gửi các event của chúng tôi đến các handler tương ứng.”*

Nếu bạn điều phối các domain event ngay trước khi commit transaction gốc, thì đó là vì bạn muốn các side-effect của các event đó được bao gồm trong cùng 1 transaction. Lấy ví dụ, nếu phương thức SaveChange của EF DbContext thất bại, transaction sẽ rollback mọi thay đổi, bao gồm cả kết quả của các thuật toán side-effect được thực thi bởi các domain event handler. Điều này là do life time của DbContext là scope, do đó đối tượng DbContext được chia sẽ trên nhiều đối tượng repository bằng việc được tạo ra bên trong cùng phạm vi(scope) hoặc object graph. Nó tương tự nhue HttpRequest scope khi phát triển các ứng dụng MVC và API.

Chính xác thì, cả 2 phương pháp (đơn nguyên tử - single atomic và tính nhất quán cuối cùng – eventual consistency) có thể đúng. Nó thực sự phụ thuộc vào domain hoặc yêu cầu nghiệp vụ của bạn và cái mà các chuyên gia domain nói với bạn. Nó cũng phụ thuộc vào khả năng mở rộng service bạn cần(nhiều các transaction chi tiết có ít tác động hơn liên quan đến các khóa database). Đồng thời nó cũng phụ thuộc vào số tài nguyên bạn sẵn sang bỏ ra vào code của mình, vì tính nhất quán cuối cùng yêu cầu code phức tạp hơn để phát hiện các mâu thuẫn có thể sảy ra trên các aggregate và nhu cầu thực hiện các hoạt động đền bù. Xem xét rằng nếu bạn commit các thay đổi tới aggregate nguyên thủy và sau đó, khi các event đang được gửi đi, nếu có 1 vân đề và các event handler không thể commit các side-effect, bạn sẽ có các mâu thẫn giữa các aggregate.

Một cách cho phép các hành động đền bù sẽ có thể lưu trữ các domain event vào trong các bảng cơ sở dữ liệu bổ sung để chúng co thể trở thành 1 phần của transaction nguyên thủy. Ngay sau đó, bạn có thể có 1 quá trình thực thi để xác đinh các mâu thuẫn và chạy các hành động đền bù bằng việc so sánh danh sách các event với trạng thái hiện tại của các aggregate. Các hành động đền bù là 1 phần của 1 chủ đề phức tạp mà cần phân tích chuyên sâu từ phía bạn, bao gồm việc thảo luận nó với người dùng nghiệp vụ và các chuyên gia domain.

Trong trường hợp nào đi nữa, bạn co thể chọn phương pháp mà bạn cần. nhưng phương pháp trì hoãn ban đầu---tạo các event trước khi commit, nên bạn sử dụng 1 transaction đơn---là phương pháp đơn giản nhất khi dung EF Core và cơ sở dữ liệu quan hệ. Nó dễ thực hiện hơn và hợp lệ trong nhiều trường hợp nghiệp vụ. Nó cũng là phương pháp được sử dụng trong ordering microservice của eShopOnContainers.

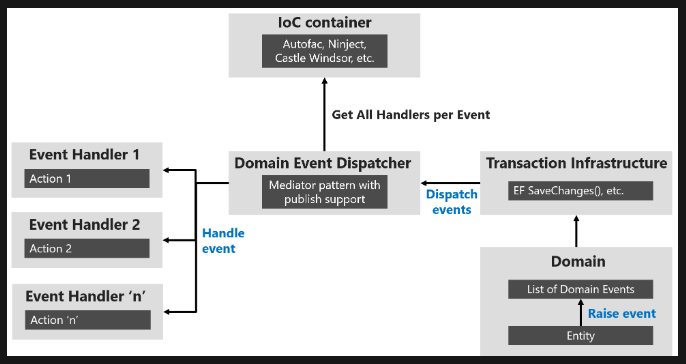
Nhưng làm cách nào để bạn gửi chính xác những event đó tới các handler tương ứng? đối tượng \_mediator bạn xem ở các ví dụ trước là gì? Nó liên quan đến các kỹ thuật và tài nguyên bạn sử dụng để ánh xạ giữa các event và event handler của chúng.

### **Domain event dispatcher: ánh xạ từ các event tới các event handler**

Một khi bạn co thể điều phối hoặc publish các event, bạn cần vài loại tài nguyên mà sẽ publish event, nên mọi handler liên quan có thể lấy nó và xử lý các side-effect dự trên event đó.

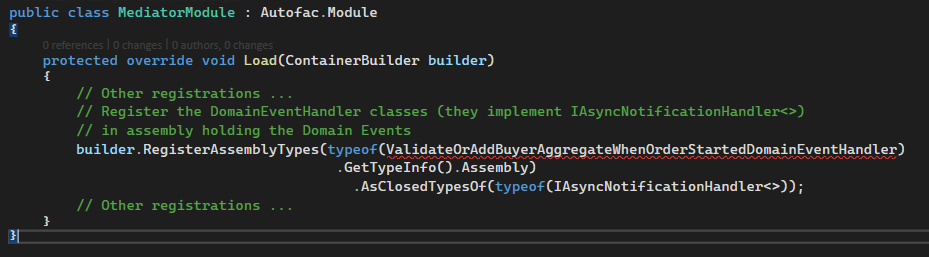
Có 1 phương pháp là hệ thông messaging thực tế hoặc với 1 event bus, có khả năng dựa trên 1 service bus chứ không phải các event trong memory. Tuy nhiên, với trường hợp đầu tiên, messaging thực sẽ quá mức cần thiết để sử lý các domain event, vì bạn chỉ cần sử lý các event đó trong cùng 1 sử lý (nghĩa là trong cùng domain và application layer).

Một cách khác để ánh xạ các event tới nhiều handler là bằng việc dùng types registration trong 1 IoC container để bạn có thể nghĩ xem nên điều phối các event đi đâu. Nói cách khác, bạn cần biết những handler nào cần cos 1 event cụ thể. Hình sau chỉ ra 1 ví dụ đơn giản về phương pháp này



Bạn có thể dụng tất cả các luồng và mô hình để tự thực thi phương pháp này. Tuy nhiên, bạn cũng có thể sử dụng các thư viện có sẵn như [MediatR](https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design). Do đó, bạn có thể trực tiếp sử dụng các interface được định nghĩa từ trước và các phương thức publish/dispatch của đôi tượng mediator.

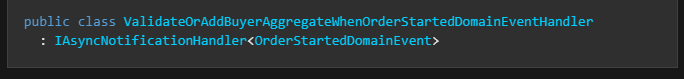
Trong đoạn code sau, đầu tiên bạn cần đăng ký các kiểu event handler trong IoC container, như ví dụ sau trong [eShopOnContainer Ordering microservice](https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.API/Infrastructure/AutofacModules/MediatorModule.cs)



Trước tiên, code xác định assembly có chứa domain event handler bằng cách định vị assembly chứa bất kỳ handler nào (sử dụng typeof (ValidateOrAddBuyerAggregateWhenXxxx), nhưng bạn có thể chọn bất event handler nào khác để định vị assembly). Vì tất cả các event handler đều triển khai interface IAsyncNotificationHandler, đoạn code đó chỉ cần tìm kiếm các type đó và đăng ký tất cả các event handler.

### **Làm thể nào để nhận các domin event**

Khi bạn sử dụng thư viện MediatR, mỗi event handler phải dùng 1 kiểu event được cung từ trên generic parameter của interface INotificationHandler, như đoạn code sau đây:



Được dựa trên mối quan hệ giữa event và event handler, có thể được coi là subscription, thư viên MediatR có thể tìm tất cả các event handler cho mỗi event và kích hoạt từng event handler đó.

### **Làm cách nào để sử lý các domain event**

Cuối cùng, eventn handler thường thực thi code tại application layer sử dụng các repository của infrastructure để lấy các aggregate bổ sung cần thiêt và thực thi logic side-effect domain. [Code event domain handler sau tại eShopOnContainer](https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.API/Application/DomainEventHandlers/OrderStartedEvent/ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler.cs), biểu thị 1 ví dụ về việc thực thi domain event:

public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler

                   : INotificationHandler<OrderStartedDomainEvent>

{

    private readonly ILoggerFactory \_logger;

    private readonly IBuyerRepository<Buyer> \_buyerRepository;

    private readonly IIdentityService \_identityService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(

        ILoggerFactory logger,

        IBuyerRepository<Buyer> buyerRepository,

        IIdentityService identityService)

    {

        // ...Parameter validations...

    }

    public async Task HandleAsync(OrderStartedDomainEvent orderStartedEvent)

    {

        var cardTypeId = (orderStartedEvent.CardTypeId != 0) ? orderStartedEvent.CardTypeId : 1;

        var userGuid = \_identityService.GetUserIdentity();

        var buyer = await \_buyerRepository.FindAsync(userGuid);

        bool buyerOriginallyExisted = (buyer == null) ? false : true;

        if (!buyerOriginallyExisted)

        {

            buyer = new Buyer(userGuid);

        }

        buyer.VerifyOrAddPaymentMethod(cardTypeId,

                                       $"Payment Method on {DateTime.UtcNow}",

                                       orderStartedEvent.CardNumber,

                                       orderStartedEvent.CardSecurityNumber,

                                       orderStartedEvent.CardHolderName,

                                       orderStartedEvent.CardExpiration,

                                       orderStartedEvent.Order.Id);

        var buyerUpdated = buyerOriginallyExisted ? \_buyerRepository.Update(buyer)

                                                                      : \_buyerRepository.Add(buyer);

        await \_buyerRepository.UnitOfWork

                .SaveEntitiesAsync();

        // Logging code using buyerUpdated info, etc.

    }

}

Code domain event handler trước được coi là code của application layer vì nó dung các infrastructure repository, như được giải thích trong phần tiêp theo về infrastructure-persistence layer. Các event handler cũng co thể ử dunngj các thành phần infrastructure khác.

### **Các domain event có thể sinh ra các integration event để đẩy ra ngoài microservice boundary**

Cuối cùng, điều quan trọng là đôi khi bạn sẽ muốn truyền đạt các event qua các microservice. Sự truyền đạt này là integration event, và no có thể được truyền đi thông qua 1 event bus từ bất kỳ domain event handler cụ thể nào.

## **Kết luận**

Như đã nêu, sử dụng cac domain event để triển khai các side-effect của các thay đổi bên trong domain của bạn. Để dùng thuật ngữ DDD, hãy ử dụng các domain event để triển khai các side-effect trên 1 hoặc nhiều aggregate. Ngoài ra, để có thể tang khả năng mở rộng tốt hơn và ít ảnh hưởng đên khóa database, hãy sử dụng tính nhất quán cuôi cùng giữa các aggregate trong cùng domain.

Ứng dụng tham triếu sử dụng MediatR để truyền đạt các domain event đồng bộ trên các aggregate, bên trong 1 transaction. Tuy nhiên, bạn cũng có thể dung vài triển khai AMQP như RabbitMQ hoặc Azure Service Bus để truyền đạt các domain event, bằng việc sử dụng tính nhắt quán cuối cùng, nhưng như đã đề cập từ trước, bạn phải xem xét sự cần thiết của các hành động đền bù trong cac trường hợp thất bại.

# Xem thêm

[Design the infrastructure persistence layer](https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/)

[Implement the infrastructure persistence layer with Entity Framework Core](https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-implementation-entity-framework-core)

[Use NoSQL databases as a persistence infrastructure](https://en.wikipedia.org/wiki/SOLID)