Thiết kế hướng nghiệp vụ với Laravel

DOMAIN ORIENTED DESIGN

Nguyễn Thế Huy

Lời mở đầu

Chào bạn, và cám ơn bạn đã đọc ebook này của mình.
Mình là Huy, hiện tại mình là Technical Leader tại Công ty Cổ phần Giao hàng Tiết kiệm. Ngoài ra, mình còn là Moderator của Group Laravel Việt Nam, một cộng đồng Laravel lớn trên Facebook với gần 30.000 thành viên.
Ebook này của mình nhằm tổng hợp các kiến thức về thiết kế ứng dụng, tập trung vào giải quyết các vấn đề nghiệp vụ (**Domain Oriented Design**), dựa trên nền tảng là Framework Laravel. Mình viết ra nó để chia sẻ quan điểm và những gì mình biết tới cộng đồng, nhằm giúp mọi người tránh được những sai lầm mình đã từng gặp phải khi thiết kế ứng dụng web. Dù bạn là người mới bắt đầu hay đã có kinh nghiệm, mình tin những kiến thức dưới đây cũng sẽ góp phần làm người đồng hành đáng tin cậy cho bạn trên con đường lập trình và thiết kế.
Chúc bạn có những giây phút thú vị, bổ ích khi đọc ebook này.

Các khái niệm cơ bản

Domain Oriented Design

Bạn không đọc nhầm đâu, là “**Domain Oriented Design**”.
Việc đưa ra khái niệm này nhằm khẳng định rằng những nội dung bạn sắp đọc dưới đây hoàn toàn không phải “**Domain Driven Design**” (DDD). Mặc dù mình vay mượn nhiều khái niệm từ DDD, nhưng ebook này hoàn toàn không phải sách hướng dẫn thiết kế DDD. Nó đúc rút những kinh nghiệm cá nhân của mình trong quá trình thiết kế ứng dụng web PHP, những đau thương mà mình muốn chia sẻ với bạn để giúp bạn tránh gặp phải. Mình mong muốn thông qua ebook này, bạn có thể thiết kế những ứng dụng hướng “nghiệp vụ” (Domain Oriented) một cách nhanh và đơn giản nhất, cũng như tìm ra một nguyên tắc lý thuyết cho chính mình.

Với cách thiết kế “Domain Oriented Design”, mình tập trung vào đối tượng “Domain”. Nó đại diện cho tất cả các vấn đề nghiệp vụ (business logic) trong ứng dụng của bạn. Thay vì thiết kế xoay quanh `Controllers / Models / Services`, mình sẽ đặt trọng tâm ứng dụng vào Domain, và tầng MVC được chuyển ra thành một Layer riêng biệt. Bạn sẽ tìm thấy thông tin này ở Section: “Refactor ứng dụng MVC của bạn”.

Ebook này thiên nhiều về PHP, nhưng mình nghĩ các ngôn ngữ lập trình hoặc Framework OOP bất kỳ cũng có thể áp dụng. Hy vọng nó sẽ đem lại thật nhiều giá trị cho bạn.

Thiết kế hướng nghiệp vụ

Thiết kế hướng nghiệp vụ là gì ?

Mình định nghĩa một ứng dụng được tổ chức "**hướng nghiệp vụ**" (hay "**Domain Oriented**") là:

- Nghiệp vụ (Business Logic) được đặt vào trung tâm ứng dụng thay vì các khái niệm kỹ thuật (Technical term).
- Bạn sử dụng được ngôn ngữ nghiệp vụ khi giao tiếp với Product Owner, Business Analyst, và sử dụng được ngôn ngữ kỹ thuật với Developer, Technical Leader một cách hoàn toàn độc lập.
- Giảm thiểu tối đa ảnh hưởng từ triển khai code hạ tầng (thay đổi framework, database, môi trường cài đặt) đến logic nghiệp vụ (Hiểu nôm ra rằng tôi có thể mang đoạn code của mình đi bất cứ đâu mà không sợ nghiệp vụ bị lỗi).
- Tốt nhất, codebase của bạn nên chính là lời mô tả cho các "User Story" của hệ thống.

Ý thứ hai có vẻ khá khó hiểu, mời bạn đến với phần tiếp theo để hiểu lý do vì sao mình cần một thiết kế hướng nghiệp vụ nhé.

Tại sao cần thiết kế theo hướng nghiệp vụ ?

Các dự án phần mềm thường có xu hướng càng ngày càng phình to do yêu cầu nghiệp vụ càng ngày càng phức tạp. Điều này là không thể tránh khỏi, vì phải đáp ứng những yêu cầu càng ngày càng cao và sự cạnh tranh trên thị trường. Hãy tưởng tượng một tình huống giao tiếp, khi bạn BA (Business Analyst) tới và đưa yêu cầu cho bạn:

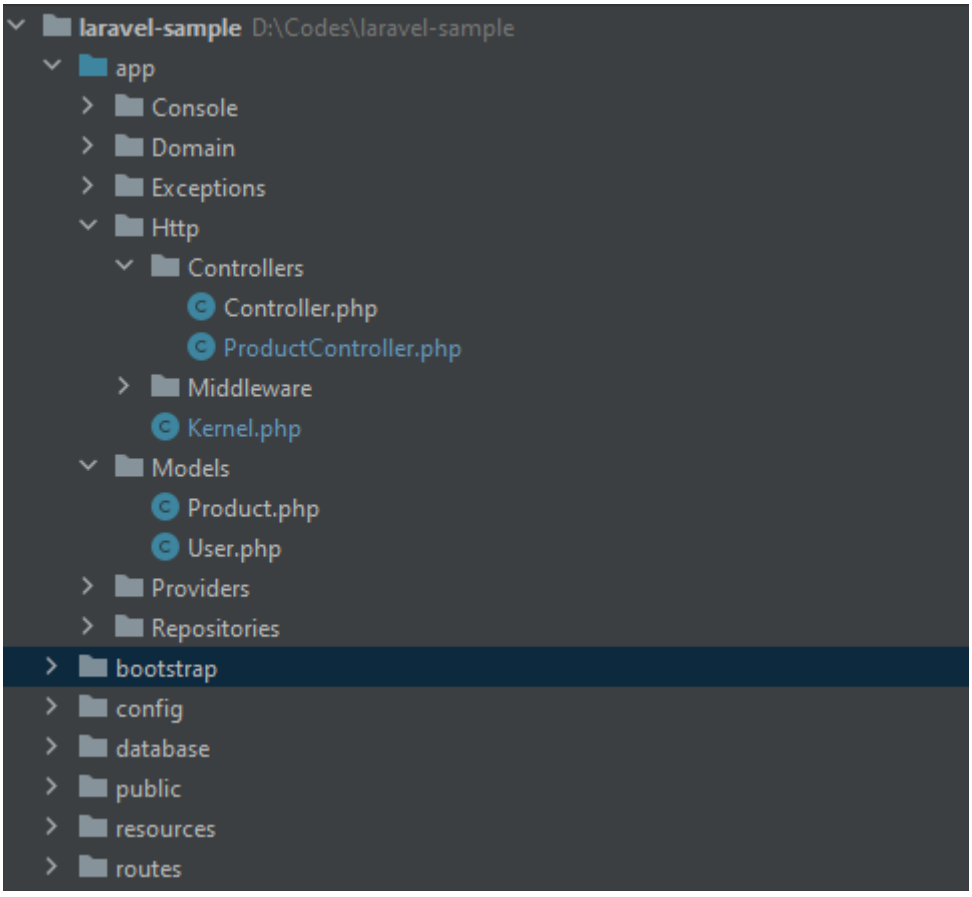
BA: Em cần sản phẩm trên website có thể support được tính năng có nhiều biến thể khác nhau, các biến thể có giá, có màu riêng biệt.

Bạn: Không vấn đề gì, anh sẽ thêm một cái bảng ABC, thêm một đoạn code XYZ và tính năng sẽ chạy trơn tru.

Điều này nghe qua có vẻ bình thường, nhưng thực tế nó sẽ phát sinh các vấn đề:

- Tư duy gắn chặt nghiệp vụ với thiết kế kỹ thuật dẫn đến sự phụ thuộc chặt chẽ giữa hai yếu tố này với nhau: việc mở rộng tính năng sẽ càng lúc càng trở nên rủi ro và khó khăn.
- Tốc độ prototype tính năng bị giảm đi đáng kể
- Vì bạn và người làm nghiệp vụ không nói chung một ngôn ngữ, nên hai bên có thể gặp tình huống miss giao tiếp, ảnh hưởng đến kết quả dự án. (Tất nhiên, chúng ta có thể có các role trong team để đảm bảo việc này được thông suốt, nhưng nếu các Developer trực tiếp có thể hiểu đúng và đủ nghiệp vụ ngay từ đầu, điều này sẽ rất có lợi cho việc phát triển sau này).

Hoặc, bạn có thể nhìn vào một ứng dụng MVC truyền thống trên Laravel:



Rất khó để đoán định Project này sẽ làm gì nếu chỉ nhìn qua các `Controllers / Models`. Thông thường, bạn sẽ phải trace từ file `routes.php` đi và lần mò dần các nghiệp vụ của dự án. Điều này, thông thường là sẽ rất khó cho người khác tiếp cận nếu dự án của bạn có nhiều logic phức tạp, chồng chéo lên nhau. Từ đó xuất phát nhu cầu có một cách triển khai dự án tập trung vào giải quyết các vấn đề nghiệp vụ, cũng như dễ dàng refactor từ mô hình MVC.

Thiết kế hướng nghiệp vụ như thế nào ?

Trong khuôn khổ của ebook này, mình mong muốn đưa đến cho bạn một cách thiết kế đơn giản nhất có thể (Chỉ cần Refactor tối thiểu từ mô hình MVC). Bạn sẽ không cần phải làm quen quá nhiều với các Pattern và khái niệm phức tạp như DDD. Các section quan trọng sẽ bao gồm:

- Làm việc với dữ liệu: Cho bạn Pattern cơ bản khi làm việc với Data trong ứng dụng.
- `Domain Models`: Đại diện cho các đối tượng nghiệp vụ, tách biệt với đối tượng cơ sở dữ liệu (hoặc đối tượng theo hướng Technical).
- `Actions`: Các business logic nghiệp vụ sẽ nằm tại Actions.
- `Repositories`: Cầu nối quan trọng giữa tầng Domain (Nghiệp vụ) đến tầng Application / Infrastructure (Ứng dụng / Hạ tầng) của bạn.

Hãy nhớ, cách thiết kế này không phải “Domain Driven Design”. Mình hy vọng bạn có thể tìm thấy một “Phương pháp luận” từ ebook này, giúp bạn có một cơ sở lý thuyết vững chắc hơn khi bạn bắt đầu thiết kế ứng dụng của mình. Việc thiết kế ứng dụng là rất linh hoạt và nó phản ánh quan điểm của mỗi người, nên hãy yên tâm nếu bạn và mình có những quan điểm khác nhau trong thiết kế.

Dependency Injection

Viết về thiết kế hướng nghiệp vụ (Business), nhưng khái niệm đầu tiên mình nói đến thì lại là Dependency Injection □. Mình muốn chia sẻ về nó đầu tiên vì trong quá trình làm việc với các bạn Junior, mình thấy các bạn còn nhiều vướng mắc khi sử dụng Dependency Injection (DI).

Hãy cùng nhắc lại một bài toán kinh điển là cách thiết kế "bóng đèn - đui đèn": Một mẫu thiết kế tốt cho phép bạn linh hoạt thay đổi giữa bóng và đui, trong khi thiết kế tồi khiến cái đèn của bạn không thể thay thế (tight coupling), khiến bạn buộc phải mua một cái đèn mới, thay vì chỉ cần thay bóng. Hãy tưởng tượng cái đèn nhà vệ sinh của mình bị hỏng, và nếu phải thay tuốt tuồn tuột từ trên xuống dưới thì thực sự là cơn ác mộng !
Thành thạo các design pattern cơ bản là điều kiện tiên quyết để bạn phát triển kỹ năng coding của mình. DI là giải pháp thiết kế cho chính bài toán bóng đèn - đui đèn bên trên. Nếu bạn làm Laravel, chắc chắn bạn đã nằm lòng khái niệm `Service Container` - một cái hộp thần kỳ giúp ứng dụng của bạn trở nên uyển chuyển một cách thú vị thông qua việc binding các lớp trừu tượng. Tuy nhiên, nếu bạn là người mới bắt đầu, hầu hết chúng ta chỉ cảm nhận DI qua việc inject interface từ `__construct`, binding trong `Service Container`. Rất nhiều bạn thắc mắc rốt cuộc DI có gì hơn việc sử dụng từ khóa new trong chính class đó (Việc inject nhiều thực sự là rối rắm, đúng không ?).

Nếu không sử dụng DI, bài toán bóng đèn của chúng ta có thể được thực thi như thế này:

```
// Class đèn
class Lamp {
    public function __construct()
    {
        // Class bóng đèn được gắn chặt vào class Đèn qua từ khóa new
        $this->bulb = new Bulb();
    }

    public function turnOn() { ... }
}
```

Về cốt lõi, DI đại diện cho triết lý "`Composition Over Inheritance`". Một đối tượng nên được tạo thành từ các viên gạch nhỏ hơn từ bên ngoài, thay vì chúng ta tự khởi tạo trong chính đối tượng đó. Hãy quay trở lại với ví dụ trên: Nếu ta đã có một cái đui, ta có thể inject bất kỳ cái đèn nào vào cái đui đó để thu được một cái đèn như ý muốn. Cần đèn xanh, có. Cần đèn vàng, có. Cần đèn nhấp nháy, easy (Đoạn này nghe quen quen phải không, DI giúp chúng ta tạo nên tính Đa hình (`Polymorphism`). Ta có thể chế ra đủ loại đèn mà mình muốn). Nếu cái đui gắn chặt vào một cái đèn (giống như nó tự khởi tạo new một cái đèn bên trong nó), rõ ràng là rất tồi tệ (ko thể thay đèn mà sẽ phải vứt nguyên mua cái mới). Khi ta tách được đèn và đui, việc của chúng ta là thiết kế một lớp trừu tượng, ở đây chính là cái trôn để xoáy đèn vào đui của nó. Cái đèn chỉ cần "implement" "interface đui đèn", và miễn là nó implement interface này, nó sẽ luôn xoáy được vào đui, và chắc chắn nó sẽ hoạt động được. Rất trực quan và dễ hiểu đúng không các bạn ?

Ví dụ giờ mình viết class Lamp bên trên như thế này:

```
class Lamp
{
    // Injection thông qua Interface Bulb, hoặc cũng có thể là Abstract Class Bulb
    public function __construct(
        Bulb $bulb
    ) {
        $this->bulb = $bulb;
  }

    public function turnOn()
    {
        $this->bulb->turnOn();
  }
}
```

Giờ nếu muốn tạo một cái đèn màu xanh, bạn có thể khởi tạo như thế này:

```
$lamp = new Lamp(new GreenBulb());
```

Một cái đèn màu đỏ, thì sẽ là:

```
$lamp = new Lamp(new RedBulb());
```

Hãy nhớ nhé: ***"Composition over Inheritance" là nguyên tắc mà các lớp phải đạt được hành vi đa hình và tái sử dụng code bằng thành phần của chúng thay vì kế thừa từ lớp cơ sở hoặc lớp cha***.

DI giúp bạn test dễ hơn nhiều. Tôi muốn test cái đui, hoặc đèn, tôi có sẵn đèn mẫu (Mocking Object), cắm vào, sáng, và thế là pass Test. Nếu không có DI, thì class sẽ rất khó để test được.
Việc áp dụng triệt để triết lý này khi code giúp chúng ta có tư duy phân tách code thành các đơn vị nhỏ, để lắp ghép nó thành những cái đèn của chúng ta. Nhiều cái đèn thì có thể làm thành một ngọn hải đăng chẳng hạn. Và nếu có cái đèn nào vỡ / hỏng (bug), ta dễ dàng cô lập, thay thế, và sửa chữa nó. Ở các section bên dưới, tư duy ưu tiên việc chia nhỏ, lắp ghép sẽ đóng vai trò rất quan trọng trong mindset thiết kế của mình.

Làm việc với dữ liệu

Làm việc với dữ liệu là một trong những công việc chính của các ứng dụng web, nếu không nói là công việc quan trọng nhất. Tuy nhiên, thao tác với dữ liệu trong PHP thực sự là một bài toán nan giải: bạn hầu như chỉ có một công cụ duy nhất là Array. Array có thể rất tiện lợi vì tính linh hoạt mạnh mẽ của nó, nhưng cũng là một điểm yếu chí mạng:

- Bạn không biết kiểu dữ liệu bên trong nó là gì: Có thể là một số nguyên, một String, nhưng cũng có thể là một cái array, một cái object …. Đương nhiên, bạn cũng rất khó tận dụng được các tính năng gợi ý của IDE trên Array, khiến việc lập trình trở nên rủi ro hơn.
- Rất khó đoán biết cấu trúc của Array: Array của bạn có thể vừa có key là Index dạng số, nhưng cũng có thể một String. Làm sao bạn biết chắc chắn một field dữ liệu nào đó chắc chắn tồn tại trong Array của mình hay không ?. Mình biết bạn có nhiều cú pháp để kiểm tra, ví dụ isset, ??, nhưng việc phải kiểm tra khắp mọi nơi sẽ khiến project của bạn trông rối rắm, và nó cũng rủi ro nữa: Chỉ một lần quên kiểm tra và bạn rất dễ gặp lỗi: Undefined Index □
- Việc đọc lại code cũ cũng trở thành ác mộng (Vì đâu ai biết được cái Array đó là gì đâu □). Và codebase của bạn sẽ càng ngày càng xuống cấp, vì người tiếp theo phải tiếp tục sử dụng và phát triển trong tình thế “không-thể-xoá-đoạn-code-nào-cả”.

Chắc chắn về dữ liệu bạn đang thao tác

Hãy cùng quan sát một ví dụ đơn giản nhé

```
$rows = $this->resultReadingFromAnExcelFile();
foreach($rows as $row) {
     $this->process($row['title'), $row['index']);
}
```

Một ví dụ khác, tình huống này rất phổ biến khi làm việc với HTTP Request

```
function store(Request $request)
{
    $data = $request->validated();

    $name =  $data['name'];
    $category = $data['category'];
    // ...
}
```

Không ai có thể chắc chắn về các Array trên, đưa bạn đến tình huống dở khóc dở cười

- dd (Dump and Die) ở mọi chỗ để biết Array đó là gì (Hoặc chạy debugger).
- Phải dò source code khắp nơi để đọc hiểu Array này (Nếu may mắn có thể bạn sẽ có documentation).
- Rủi ro truy cập các index không tồn tại, hoặc kiểu dữ liệu không phù hợp.
- Nếu có ai đó sửa các Array này ở phía trước thì đúng là thảm hoạ.

Các ngôn ngữ sở hữu Strong Type System thường sẽ có các cơ chế giúp bạn giải quyết vấn đề này. Không may lắm là PHP chưa sở hữu một hệ thống tương tự. Bạn chỉ có trong tay Array và Object mà thôi. Section bên dưới về Data Transfer Object mình sẽ bàn luận sâu thêm về việc giải quyết vấn đề này.

Data Transfer Objects

Ở bên trên, bạn đã biết được việc chỉ sử dụng Array để luân chuyển dữ liệu trong ứng dụng của mình đem lại nhiều bất lợi. Một điều quan trọng nữa trong ứng dụng của bạn, dữ liệu có thể tới từ nhiều nguồn khác nhau. Hãy tưởng tượng nghiệp vụ “Import” sản phẩm của bạn có thể đến từ một lời gọi HTTP, cũng có thể thông quan một file Excel, hoặc một Event đến từ một hệ thống Service nào đó. Sẽ rất khó để tái sử dụng Code nếu phần xử lý của bạn phụ thuộc trực tiếp vào một HTTP Request hoặc một lệnh đọc file như thế này:

```
function doSomething(array $product)
{
    $product[/* Xử lý array này như thế nào đây?? */];
}
```

Và đây là lúc Data Transfer Objects (DTO) xuất hiện:

```
class Product
{
    /** @var string */
    public string $title;

    /** @var Category*/
    public Category $category;

    /** @var bool */
    public boolean $active;
 }
```

Refactor một chút, chúng ta sẽ có đoạn code như sau:

```
function doSomething(Product $product)
{
    // Chính Object Product đã giúp mô tả chính nó
    $product->category->name; // IDE có thể dễ dàng gợi ý đoạn code này chuẩn xác
    $product->title; // Bạn chắc chắn biết đây là một string
}
```

Bạn có thể dễ dàng khởi tạo một DTO thông qua Constructor, Getter, Setter, hoặc một static function:

```
$product = new Product([
    'title' =>  'Đây là tên sản phẩm',
    'active' => false,
    'category' => new Category()
]);
```

Những điểm lợi thế quan trọng mà DTO mang tới cho codebase của bạn:

- Bạn biết chính xác và hoàn toàn tự tin về dữ liệu mình đang sử dụng.
- Type hint giúp việc code rõ ràng chính xác hơn.
- Với DTO, bạn hoàn toàn không cần quan tâm đến việc tương tác đầu vào. Giờ đây bạn có thể dễ dàng tái sử dụng code của mình theo nhiều nghiệp vụ khác nhau.
- Các DTO rõ ràng cũng đóng vai trò mô tả cho user story, giúp tăng tính “hướng nghiệp vụ” cho codebase của bạn.

Repositories

Trong các cuộc phỏng vấn mình tham gia, hầu hết mọi người đều sử dụng Repository trong dự án, và mục đích sử dụng thường là "nhằm linh hoạt thay đổi database và tái sử dụng code". Câu hỏi đặt ra là:
Có bao nhiêu dự án bạn từng làm thực sự thay đổi database hoàn toàn thông qua Repository ?
Câu trả lời: "Rất hiếm khi". Việc thay đổi database bên dưới (từ Mysql sang MongoDB chẳng hạn) đòi hỏi chi phí chuyển đổi to đùng hơn là chỉ thay đổi cách implement interface. Thông thường các db sẽ bổ trợ nhau, chứ hiếm khi thay đổi hoàn toàn.Nếu để tái sử dụng code, thì câu hỏi đặt ra là:
Có nhất thiết phải tạo một lớp trừu tượng (interface) rồi implement lại như Repository ?. Rõ ràng có thể sử dụng Service Layer để tái sử dụng các phương thức đó (Và tất nhiên rồi, chúng ta sử dụng Eloquent ở đây phải không ?)
Ví dụ: hàm `findByABC()` => bên trong là `Model::query()->where('abc', 'something')->get()` hoàn toàn có thể được triển khai ở lớp Service layer và vẫn không thay đổi gì (Nếu mình không gọi nó là Repository nữa thì cũng ko ai nhận ra cả). Rồi, thế rốt cục là Repository nên được hiểu như thế nào ? Nói như vậy thì chả lẽ bỏ luôn Repository à ? Các bạn bình tĩnh đọc tiếp nhé.
"Repository Pattern" được định nghĩa trong Patterns of Enterprise Application Architecture (của Martin Fowler, các bạn nhớ phải kiếm cuốn này đọc nha) rằng nó:
Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects
Oke, hiểu như thế này: Repository nên là một lớp trung gian giữa business logic và lớp hạ tầng (triển khai Database hoặc giao tiếp với external service => Eloquent là một ví dụ tiêu biểu cho triển khai giao tiếp DB lớp hạ tầng. Và đương nhiên bạn có thay thế Eloquent bằng khẩu vị của bạn (Doctrine chẳng hạn))
Hãy lấy một ví dụ: Bạn đang làm một dự án có đối tượng sản phẩm (Product). Tại layer nghiệp vụ (Domain) của bạn, bạn định nghĩa một đối tượng là Product. Hãy nhớ đoạn này, và bạn cũng đừng nhầm lẫn, đối tượng Product này khác hoàn toàn với Model Product của Laravel (thứ đại diện để tương tác với database).
Mình sẽ triển khai một interface `ProductRepository` như thế này:

```
interface ProductRepository {
    public function get($id): Product;
    public function save(Product $product): void;
    public function delete(Product $product): void;
}
```

Interface này mình sẽ đặt nó ở layer Domain (Nghiệp vụ). Lớp Domain hoàn toàn giao tiếp nội bộ bên trong nó, và nó "mù" hoàn toàn về việc triển khai (implement) lại interface này. (tức là nó sẽ không cần cách implement ở khía cạnh technical, rất chuẩn phong cách làm nghiệp vụ □).

Đó là lý do vì sao trong meetup group Laravel tại Hà Nội, chúng mình nhắc đến việc "Không sử dụng Eloquent". Clear hơn một chút, mình "không" dùng Eloquent trong lớp nghiệp vụ, mà đẩy nó xuống lớp hạ tầng (Infrastructure) mà thôi. Điều này đem lại lợi ích quan trọng cho việc:

- Nghiệp vụ được rõ ràng, tách bạch khỏi technical design. Đối tượng Product của bạn kia có thể được đại diện bởi 3-4 bảng ở dưới database (và có thể bị thay đổi thường xuyên). Bóc tách thành 2 phần riêng biệt giúp bạn linh hoạt hơn nhiều trong thiết kế khi nghiệp vụ phức tạp.
- Linh hoạt trong triển khai: Yes, bạn không nghe nhầm đâu, Repository ngoài đại diện tương tác database, còn có thể được implement để gọi API (chuyện bình thường, như call ElasticSearch chẳng hạn). Việc bạn nâng cấp hạ tầng cũng không hề ảnh hưởng tới nghiệp vụ đã triển khai.

Nếu bạn vẫn cảm thấy khó hiểu, thì mình có thể tóm tắt:

Repository không phải ORM, đừng dùng nó như ORM (Vì như thế dùng thẳng ORM còn hơn). Repository tách biệt hoàn toàn khỏi Technical design, vì nó sẽ là cầu nối giữa nghiệp vụ và triển khai hệ thống.

Nhưng dự án của bạn khá đơn giản thì sao ?. Theo mình, nếu dự án nhỏ, thì không nên sử dụng Repository. Hãy tận dụng những tính năng mạnh mẽ của Laravel Model như Scope, Custom Query Builder, Macro... Nó cực kỳ linh hoạt và bạn sẽ không phải máy móc tạo interface mà không hiểu rốt cục nó để làm gì cả. Mình sẽ bàn luận về Custom Query Builder của Laravel ở section tiếp theo nhé

Custom Query Builders

Section này xuất hiện là một điều thú vị. “`Custom Query Builders`” là khái niệm riêng của Framework Laravel, và nếu đúng ra, thì chúng ta nên hạn chế phụ thuộc vào hạ tầng Framework. Tuy nhiên, các “`Custom Query Builders`” giúp chúng ta đáng kể trong việc chia tách các luồng làm việc với dữ liệu, và sẽ rất phù hợp nếu bài toán của bạn không nhất thiết cần tới các lớp Repositories cồng kềnh.

Hãy cùng quan sát các ví dụ sau nhé:

```php
namespace Domain\Products\QueryBuilders;

use Illuminate\Database\Eloquent\Builder;

class ProductQueryBuilder extends Builder
{
    public function whereActivated(): self
    {
        return $this->where('is_active', true);
    }
}

// Ở trong model

class Product extends Model
{
    public function newEloquentBuilder($query): ProductQueryBuilder
    {
        return new ProductQueryBuilder($query);
    }
}
```

Với ví dụ trên, mình đã đẩy các phương thức tương tác lấy dữ liệu ra khỏi model, và đưa nó sang lớp QueryBuilder. Tất nhiên, với ví dụ trên, bạn có thể sử dụng Scope luôn trong Model, nhưng đõ rõ ràng không phải cách để scale sau này.

Giờ đây bạn hoàn toàn có thể làm như thế này:

```
return Product::query()->whereActivated();
```

Bạn cũng hoàn toàn có thể áp dụng nó với Model Collection, thông qua phương thức `newCollection`:

```
namespace Domain\Products\Collections;

use Domain\Products\Models\InvoiceLines;
use Illuminate\Database\Eloquent\Collection;

class ProductCollection extends Collection
{
    public function isActive(): self
    {
        return $this->filter(function (Product $product) {
            return $product->isActive();
        });
    }
}

// Ở trong Model

class Product extends Model
{
    public function newCollection(array $models = []): ProductCollection
    {
        return new ProductCollection($models);
    }

    public function isActive(): bool
    {
        return $this->is_active;
    }
}
```

Và bạn có thể sử dụng nó như thế này:

```
return Product::query()->isActive()->get()->toArray();
```

Bạn thấy đó, với Custom Query Builders, chúng ta vẫn chia tách được tầng tương tác dữ liệu hợp lý. Hãy cân nhắc sử dụng nó trong tình huống phù hợp nhé.

ViewModels

ViewModels là một cách tốt để bạn phân chia logic xử lý ở tầng View. Khi nhắc đến View, hãy hiểu là mình không chỉ nhắc tới HTML hay các Template Engine như Blade, mà còn là dữ liệu cho các API dạng JSON, XML nữa. Để đơn giản, bạn hãy tưởng tượng view models là những class xử lý dữ liệu cho view, hơn là đẩy nó xuống các layer như Actions, Repositories. Điều này giúp code của bạn tổ chức tốt hơn, tái sử dụng và linh hoạt trước sự thay đổi của nghiệp vụ.

Hãy bắt đầu với một ví dụ đơn giản. Bạn cần làm một Form tạo sản phẩm, với các danh mục được load ra từ database. Controller của bạn sẽ trông như thế này:

```
public function create()
{
    return view(product.create, [
        'categories' => Category::all(),
    ]);
}
```

Chúng ta sẽ cần thêm một cái form nữa cho sửa sản phẩm, và với case sửa thì bạn sẽ cần thêm đối tượng sản phẩm cho view. OK, controller của bạn sẽ có method edit như thế này:

```
public function edit(Request $request)
{
    // Mình xin phép bỏ đi các đoạn validate nhé
    // Mình tái sử dụng lại view create ở bên trên, tạm thời sẽ dùng tên này cho mọi người dễ tưởng tượng nhé
    $product = Product::find($request->id);
    return view(product.create, [
        'product' => $product,
        'categories' => Category::all(),
    ]);
}
```

Sẽ như thế nào nếu bạn cần scale nghiệp vụ, thay đổi các dữ liệu hoặc điều kiện trả ra view ?. Ví dụ trả thêm tags, labels ?. Ví dụ bạn cần giới hạn theo quyền của User hoặc theo đối tượng Product ?. Bạn sẽ

phải sửa tất cả những chỗ bạn cần render ra View ? (Hoặc render ra JSON). Lúc đó bạn sẽ phải tìm hết các hàm trên để bổ sung lại code sang như thế này:

```
return view(product.create, [
    'categories' => Category::all(),
    'tags' => Tag::all(),
]);
```

Thêm nữa, việc không thống nhất những gì ở View sẽ khiến bạn lúng túng và bối rối: Liệu biến hay phương thức này có tồn tại ở View không, tôi có đang bỏ lỡ điều gì không ?. Điều này thực sự không tốt tí nào, và đây là lúc bạn sẽ cần tới ViewModels.
View model đóng gói tất cả các logic có thể tái sử dụng, với một chức năng duy nhất: Chuẩn bị dữ liệu chính xác cho View.
Hãy cũng xem class ProductViewModel dưới đây:

```
class ProductViewModel
{
    public function __construct(
            Product $product = null
    ) {
        $this->product = $product;
    }

    public function product(): Product
    {
        return $this->post ?? new Product();
    }

    public function categories(): Collection
    {
        return Category::all();
    }
}
```

Những đặc điểm quan trọng giúp ViewModel phát huy vai trò trong Project của bạn:

- Sử dụng tối đa Dependency Injection, cho bạn lợi thế flexible khi có thể điều chỉnh code từ bên ngoài. Tất nhiên là cả lợi thế về Test nữa.
- Chuẩn hoá các phương thức để bạn có thể sử dụng ở View. Tất nhiên rồi, bạn không cần đoán các biến hay phương thức nào sẽ được trả ra ở View nữa.
- Đóng gói và tái sử dụng triệt để.

Controller của bạn giờ sẽ trông như thế này:

```
class ProductController
{
    public function create()
    {
        $productViewModel = new ProductViewModel();

        return view('product.create', compact('productViewModel'));
    }

    public function edit(Request $request)
    {
        $product = Product::find($request->id);
        $productViewModel = new ProductViewModel($post);

        return view('product.create', compact('productViewModel'));
    }
}
```

Và đương nhiên, bạn có thể viết View một cách “an toàn” hơn như thế này:

```
<h1>{{ $productViewModel->product()->name }}</h1>
<select>
    @foreach ($productViewModel->categories() as $category)
        <option value="{{ $category->id }}">
            {{ $category->name }}
        </option>
    @endforeach
</select>
```

Tóm lại, `ViewModel` là một cách khá hay khi bạn cần làm việc với dữ liệu trong `View` và `Controller`. Chúng cho phép tái sử dụng tốt hơn và đóng gói logic, thứ vốn không nên được thể hiện trong `Controller`.

Domain Models

Mình tách Domain Model ra một phần riêng chứ không đưa vào section “Làm việc với dữ liệu”, vì cảm thấy đây là một phần rất quan trọng trong Domain Oriented Design.

Domain Models hay các Entity đại diện cho các thực thể trong nghiệp vụ của bạn. Về mặt “Vật lý”, nó là các PHP Plain Object (POJO), không kế thừa hay triển khai lại bất cứ class hay interface nào của Framework. Chúng có định danh (Identification) rõ ràng, và được phân biệt với nhau thông qua các định danh này (Tức là nếu chung ID, thì ta có hai Entity là cùng trỏ vào một thực thể nghiệp vụ, chứ không cần so sánh các thuộc tính khác).

Các Models này thuần tuý là các bản mô tả nghiệp vụ, và bạn sẽ dễ dàng nhìn thấy logic của ứng dụng thông qua chúng. Hãy cùng lấy ví dụ sau về Domain Model:

Đối tượng sản phẩm trong một website Ecommerce thông thường:

```
// ProductEntity.php (Domain Layer)
namespace Domain;

class ProductEntity {
    private $id;
    private $name;
    private $price;
    private $remain;

    public function __construct($id, $name, $price) {
        $this->id = $id;
        $this->name = $name;
        $this->price = $price;
    }

    public function getId() {
        return $this->id;
    }

    public function getName() {
        return $this->name;
    }
```

```
    public function getPrice() {
        return $this->price;
    }

    public function getRemain() {
        return $this->someActionToGetRemainProduct->handle();
    }
}
```

Thực tế triển khai trên Database đối tượng sản phẩm sẽ bao gồm nhiều hơn một Model

```
class Product extends Model
{
   // Các thuộc tính của sản phẩm
}

class ProductWarehouse extends Model
{
  // Thông tin sản phẩm được lưu trong kho
}
```

Khi làm việc với nghiệp vụ, bạn chỉ tương tác với `ProductEntity` theo cách:

```
return $productEntity->getRemain(); // Lấy ra số sản phẩm tồn kho.
```

Bạn thấy điểm khác biệt rồi chứ ?. Đây là điều quan trọng khi thiết kế, vì chúng ta đang sử dụng một Model riêng cho nghiệp vụ, thay vì nhìn theo khía cạnh kỹ thuật (Model Database). Bạn có thể nhanh chóng prototype các thuộc tính hoặc bài toán business thậm chí trước cả khi Database được thiết kế và triển khai. Ví dụ thêm một phương thức isSomething vào Product Entity chẳng hạn. Và giả sử là bạn muốn thay đổi cách thiết kế từ sử dụng Database sang một lời gọi HTTP API, điều đó cũng không ảnh hưởng gì đến cách nghiệp vụ của bạn vận hành cả. Điều này sẽ giúp ích rất nhiều khi ứng dụng scale nghiệp vụ và có thêm Developer join dự án (Một bên prototype nhanh và một bên có thời gian để nghĩ sâu về triển khai Technical).

Services

Cách thiết kế phổ biến nhất mình thấy khi làm việc với Laravel là mô hình MVC + Service. Service thường được thiết kế để gom các business logic lại, để ae không làm cho `Controller / Model` trở thành "hố đen vũ trụ". Ví dụ chúng ta sẽ có:
Controller: `ProductController`
Model: `Product`
Service: `ProductService`

Mô hình thiết kế này bộc lộ một số nhược điểm khi ứng dụng của chúng ta lớn dần lên:

- Không thể biết use-case trên hệ thống là gì. Về cơ bản thì ta biết là project có `ProductService`, còn muốn biết nó làm gì thì thường là sẽ phải mò vào file route, đọc code, hoặc đặt debug. Về cơ bản thì nếu business logic phức tạp thì sẽ khá oải.
- Khó mà đảm bảo SOLID: `Controller` của chúng ta thường được inject nguyên các file `Service` vào để xử lý. Nếu quản lý các file `Service` không tốt, thì dễ dẫn đến việc phụ thuộc chặt chẽ vào nhau. Nó làm tăng khả năng gây ra bug nếu có nhiều người cùng làm việc trên các file `Service` lớn này.

Domains

Một ứng dụng lớn thường đặc trưng bởi những logic phức tạp, chồng chéo. Section Domains cung cấp cho bạn cách tổ chức codebase cho ứng dụng lớn một cách rõ ràng, mạch lạc, dễ bảo trì nhất có thể. Nó sẽ bao gồm:

- Use case: khái niệm và lý do bạn cần thiết kế xoay quanh các Use case.
- Actions: Cách bố trí tổ chức class nghiệp vụ trong Domain hiệu quả.

Use Case

"Use case" là gì ?

Có nhiều cách để diễn giải khái niệm này. Nếu các bạn đã làm quen với Clean Architecture, thì Uncle Bob đưa ra khái niệm về "use case" là:
A piece of business logic that represents a single task that the system needs to perform
Ở đây, mình tạm định nghĩa "use case" trong ứng dụng là một nhiệm vụ logic riêng biệt. Các "use case" cần độc lập và tách biệt nhau hoàn toàn, và tốt nhất thì nó chỉ nên làm một và một chỉ một công việc mà thôi. Đoạn này nghe giống Single Responsibility trong SOLID nhỉ 😀. Đoán xem ai là người ủng hộ mạnh mẽ nhất các principle này nào => chính là Uncle Bob 😀.

Tại sao ta nên thiết kế ứng dụng của mình xoay quanh “Use case” ?

- Đảm bảo rằng code của bạn có thể dễ dàng tái sử dụng: Các thành phần trong một "use case" có thể dễ dàng được "lắp ghép" lại để phục vụ các bài toán khác nhau, thay vì phải lặp code hoặc inject các class ngày càng cồng kềnh. Mình biết là việc maintain các class vài chục k dòng cũng là việc thường ngày ở huyện, nhưng mình vẫn thích những thứ gọn nhẹ, dễ dàng thay đổi hơn 😀. Điều này cũng đảm bảo tinh thần: Composition over Inheritance.
- Single Responsibility: Đương nhiên rồi, một nguyên tắc quan trọng của Lập trình hướng đối tượng.
- Dễ dàng quan sát nghiệp vụ của ứng dụng: Với việc gom nhóm các hành động, "use case" cho phép bạn nhanh chóng nắm bắt logic hơn là phải tập trung vào các khái niệm technical như "Controller", "Model"..
- Testable: Các use case với nhiệm vụ rõ ràng input / output độc lập, sẽ rất dễ cho việc viết test.

Actions

Bạn đã biết về khái niệm “Use case” bên trên. Mình thì lại không thích cái tên này lắm, vì cảm giác nó dễ gây nhầm lẫn. Từ giờ, mình sẽ gọi các “Use case” là “Actions”, cho nó phù hợp với ngữ cảnh, cũng như hợp trend trong giới làm Laravel (Được nhiều KoL trong giới sử dụng).

Trái tim của ứng dụng chúng ta sẽ đặt ở “`Actions`”. `Action` sẽ là một Class thực hiện một nghiệp vụ và chỉ một nghiệp vụ cụ thể trong Domain của chúng ta. Một “`Action`” cơ bản sẽ trông như thế này:

```
class CreateProductAction
{
    public function __construct(
        private readonly ImportProductWarehouseAction $importProductWarehouseAction,
     ) {
    }
    public function handle() {
        /** Thực hiện nghiệp vụ tạo sản phẩm**/
        /** Gọi sang Action Nhập kho sản phẩm **/
       $this->importProductWarehouseAction->handle();
        /** Kích hoạt nghiệp vụ sau khi tạo sản phẩm thành công**/
       event(new AfterCreateProduct);
    }
}
```

Một class với một hàm chính duy nhất (Mình gọi nó là “`handle`” để đồng nhất với convention của Laravel, bạn có thể viết nó dưới dạng một `Invokable Controller` qua magic method `__invoke()`, nhưng mình không thích cách này lắm).
Những lợi thế rất lớn của các class “`Actions`”

- Single responsibility: Việc quan trọng phải nhắc lại nhiều lần. Các class Action của chúng ta chỉ làm một và một việc duy nhất.
- Bởi vì một `Action` chỉ làm một việc duy nhất, nó trở nên cực kỳ linh hoạt: Một Action có thể được gọi bởi một `Action` khác, có thể được kết hợp lại với nhiều Action khác để thực thi một nghiệp vụ lớn hơn.
- Bạn có thể đưa các Action này sang `Queueable` dễ dàng với Laravel.
- `Action` miêu tả User story: Chính xác, việc quan sát các giúp bạn hiểu nghiệp vụ một cách nhanh chóng và hiệu quả, mang codebase của bạn đến gần hơn với ngôn ngữ nghiệp vụ. Hãy nhìn vào một project được thiết kế xoay quanh các `Action`:

```
// app/Domain/Products/
|── Actions
    ├── CreateProductAction
    ├── EditProductAction
    ├── ImportProductAction
```

Nếu một Developer mới join team, họ dễ dàng biết rằng ứng dụng đang có các tính năng

- Tạo mới sản phẩm
- Sửa sản phẩm
- Import sản phẩm

mà không cần phải đi qua những class `Controllers`, `Models`, `Services` khổng lồ. Quá tuyệt vời phải không □

Một điều nho nhỏ nữa, việc tái sử dụng các Actions hoàn toàn không cần sử dụng đến các phương thức static mà tận dụng tối đa `Dependency Injection`. Nếu bạn muốn đọc thêm về DI, mời bạn tham khảo bài viết của mình trên blog Laravel Việt Nam tại đây nhé.

Command Bus Design Pattern

Ở section `Services`, mình có nêu ra một số vấn đề khi thiết kế theo mô hình `MVC + Services`. Vậy có cách nào đơn giản để giải quyết vấn đề này không ?. Project của tôi đã được triển khai như thế này, tôi cũng không có (không muốn) phải refactor nó quá nhiều hoặc áp dụng kiến trúc quá phức tạp. Câu trả lời khả dĩ là áp dụng `CommandBus Design Pattern`. Thay vì thiết kế logic trong các service, tôi sẽ chia logic của mình thành:

- `Command`: Một class dùng để wrap dữ liệu, yêu cầu cần thiết để thực thi một logic trên hệ thống (`Data Transfer Object`). Lấy ví dụ nếu action là tạo sản phẩm, tôi sẽ tạo `CreateProductCommand`, trong đó sẽ có các thuộc tính name được gán từ request gửi lên.
- Handler: Class để xử lý duy nhất một logic cụ thể. Đầu vào sẽ là command bên trên. Ví dụ `CreateProductHandler` sẽ nhận đầu vào là `CreateProductCommand`. Trong Handler có thể gọi Repository để save vào DB, hoặc gọi các Handler khác cho các action khác bên trong.
- `CommandBus`: Luồng định tuyến, ghép `Command-Handler`, và decorate thêm các phần chúng ta cần. Lấy ví dụ ta muốn bọc các `Handler` trong một Transaction, ta có thể dựa vào CommandBus

Đoạn code sau sẽ giúp bạn cảm thấy dễ hiểu hơn:

```
//CreateProductCommand
//Command có thể đóng vai trò như DTO và là đầu vào duy nhất cho Handler.
class CreateProductCommand
{
   private string $name;
   /**
    * @return string
    */
   public function getName(): string
   {
       return $this->name;
   }

   /**
    * @param string $name
    */
   public function setName(string  $name): void
   {
       $this->name = $name;
   }
}
```

```php
//CreateProductCommandFactory
//Class đóng vai trò khởi tạo Command từ đối tượng Request
class CreateProductCommandFactory
{
   public static function make(CreateProductRequest $request): CreateProductCommand
   {
       $command = new CreateProductCommand();
       $command->setName($request->get('name'));

       return $command;
   }
}


// ProductController
public function createProduct(CreateProductRequest $request): JsonResponse
{
   // Khởi tạo đối tượng Command từ Request.
   $command = CreateProductCommandFactory::make($request);
   $handler = app()->make(CreateProductHandler::class);
   try {

       $result = Bus::dispatchNow($command, $handler);
       return Response::success($result);
   } catch (Throwable $exception) {
       return Response::error($exception->getMessage());
   }
}

//CreateProductHandler
class CreateProductHandler
{
public function __construct(
   private readonly CreateProductAction createProductAction;
) {
}
```

```
    public function handle(CreateProductCommand $command): void
    {
     // Truy cập các thuộc tính của Command
     $command->getName();
     // Tiến hành nghiệp vụ tạo sản phẩm.
     $this->createProductAction->handle();
    }
}
```

Việc áp dụng `CommandBus` vào Project không làm hệ thống của các bạn phải thay đổi quá nhiều, nhưng đem lại những ích lợi khá lớn:

- Chỉ rõ các use-case trên hệ thống
- Chia nhỏ logic, dễ dàng debug, tái sử dụng code
- Có thể dễ dàng chuyển đổi Handler trực tiếp sang Queue, Transaction nhanh chóng.

Laravel đã support sẵn `CommandBus` thông qua Facade Bus, ngoài ra thì có tactician của PHP League là một thư viện `CommandBus` rất nổi tiếng và dễ áp dụng trong Project Laravel. Nếu mọi người thấy quen, thì Event - Listener của Laravel cũng rất gần với cách thiết kế này. Về cơ bản, mình sẽ chuyển đổi Request thành các Command, và Service thành các Handler, các bạn có thể áp dụng dễ dàng ngay vào dự án.

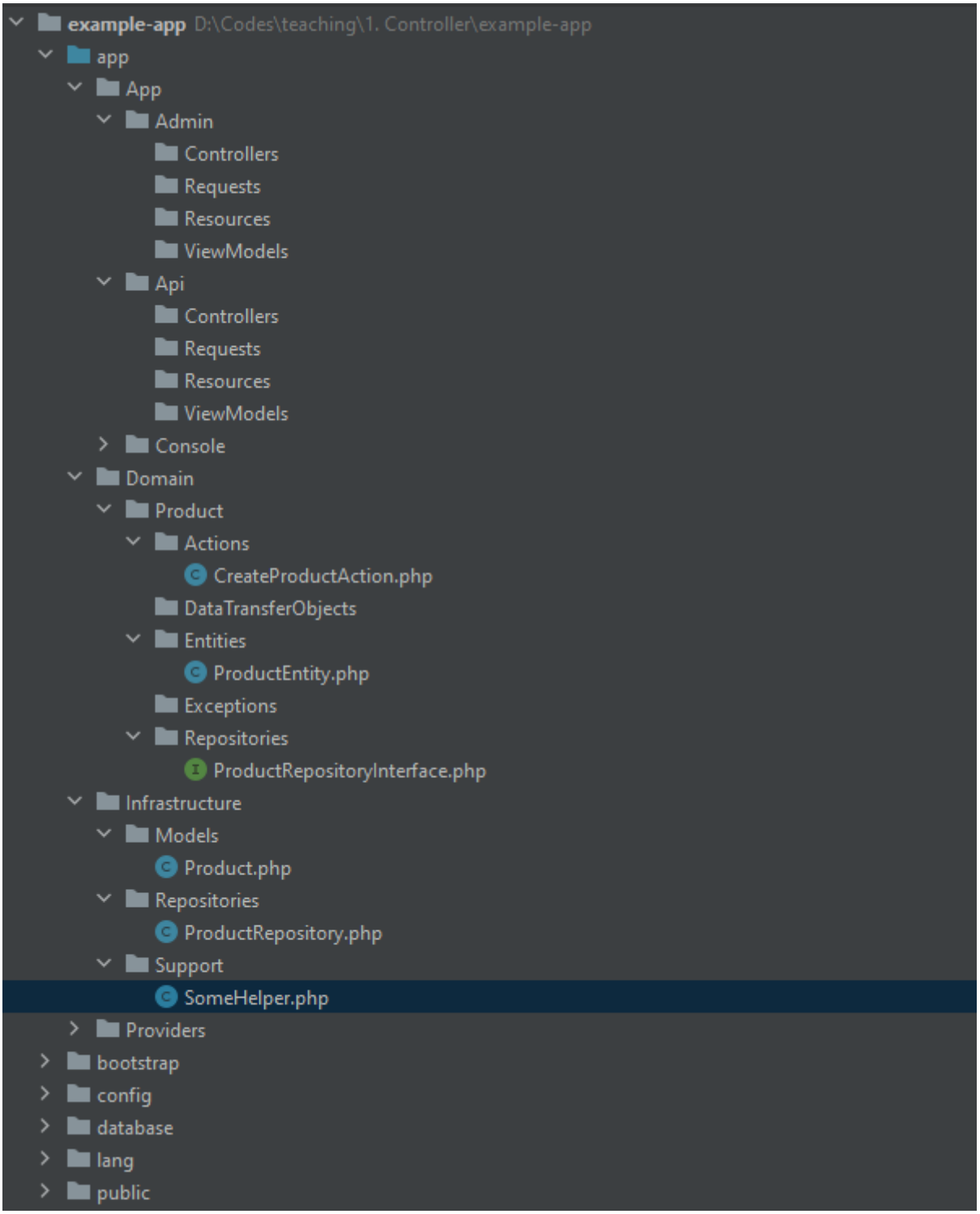
Refactor ứng dụng MVC của bạn

Một trong số những mục tiêu mình đặt ra là cần một cách Refactor đơn giản nhất cho các hệ thống MVC sẵn có. Để làm được điều này, mình sẽ giữ nguyên mô hình MVC mặc định của Laravel, và sẽ bổ sung thêm lớp Domain. Phần MVC mặc định, mình gọi nó là lớp Application hay App. Vậy chúng ta có thể có các layer sau:

- Application: Mô hình MVC mà bạn đã quen thuộc
- Domain: Tất cả các class liên quan tới nghiệp vụ: Repository Interface, Domain Model (Entities), Actions, DTO, ….
- Infrastructure: Các thành phần tương tác trực tiếp với Framework, hạ tầng như kết nối đến Database.
- Support: Phần tương tác với các ứng dụng bên thứ ba, các helper dùng chung.

Một lần nữa, bạn sẽ thấy nó khác biệt với DDD. Cách thiết kế này có thể chưa hoàn toàn triệt để tách biệt hạ tầng và nghiệp vụ, nhưng nó đảm bảo tới 90% vấn đề này, và giúp bạn refactor ứng dụng rất nhanh. Trên thực tế, bọn mình đã maintain các ứng dụng tương đối lớn (hàng trăm nghìn line of codes) theo mô hình này tương đối ổn.

Hãy cùng quan sát Structure sau nhé:



Bạn thấy đó, nó vẫn là những gì bạn vốn dĩ đã rất quen thuộc với Laravel. Một chút custom và chúng ta có một cách thiết kế cụ thể và rõ ràng mạch lạc hơn. Nếu bạn là người khó tính, bạn có thể đặt lại toàn bộ cấu trúc mới sang một thư mục khác tách biệt hoàn toàn với skeleton của Laravel. Nhưng như mình đã nói, mình thích một giải pháp gọn nhẹ và “mỳ ăn liền” hơn □.

Lời kết

Bạn đã đọc hết cuốn “Thiết kế hướng nghiệp vụ với Laravel” của mình.
Từ đáy lòng, mình cám ơn bạn rất nhiều vì đã đọc đến cuối quyển Ebook này. Có thể cách viết mình còn lộn xộn, mình mong bạn hiểu những tâm huyết của mình, và những mong mỏi được chia sẻ kiến thức đến cộng đồng. Thiết kế ứng dụng là một câu chuyện dài, mình chỉ hy vọng rằng những gì mình viết ra sẽ giúp bạn phần nào trên con đường lập trình đầy chông gai và thú vị.

Cám ơn bạn rất nhiều !
Nếu bạn thích những gì trong cuốn sách này và muốn trao đổi thêm, đừng ngần ngại liên lạc với mình qua:
Email: huynt57@gmail.com
Facebook: Tại đây

Cuốn Ebook này sẽ không tồn tại nếu không có vợ mình Dương Thị Thu Huyền và con trai mình, cố vấn tí hon Nguyễn Dương Hoàng Khôi. Cám ơn gia đình đã luôn bên cạnh và ủng hộ mình.

Happy Coding !