Organizing code into domain modules

26 November 2017 - Matthieu Napoli

Note: the french version of this article is published on Wizaplace's tech blog.

We recently discussed 2 topics seemingly unrelated with my colleagues at Wizaplace:

- how to organize code?
- how to organize teams?

Feature teams

Regarding "how to organize teams", we were discussing Spotify's *Feature teams*. In a "classic" organization, teams are usually formed by grouping people based on their job title:

- backend developers team
- frontend developers team
- sysadmin team
- product team

But in a "feature team" organization, teams are organized... by features:

- team *e-commerce*: 1 product owner, 3 developers, 1 sysadmin
- team *blog*: 1 product owner, 2 developers, 1 sysadmin

The pros of this kind of organization are numerous and I do not intend to go over them here. But what does this have to do with code organization?

The classic code structure

Most projects will follow this "classical" layout:

```
src/
    Entity/
        Product.php
        Basket.php
    Service/
```

```
        ProductService.php
        BasketService.php
    Repository/
        ProductRepository.php
        BasketRepository.php
    Resources/
        ORM/
            Product.orm.yml
            Basket.orm.yml
```

Code is organized by type, i.e. what each thing **is** (an entity, a service, a repository, ...).

A developer *is* a developer, a sysadmin is a sysadmin, but when they both work on the same project, it makes sense to let them be in the same team. How about doing the same thing with code? Why not group code based on what it *does* instead of what it *is*?

Module-oriented code structure

Here is an alternative layout where code is grouped by "domain module", or feature:

```
src/
    BasketModule/
        Basket.php
        BasketService.php
        BasketRepository.php
```

```
        ORM/
            Basket.orm.yml
    ProductModule/
        Product.php
        ProductService.php
        ProductRepository.php
        ORM/
            Product.orm.yml
```

The first reaction one could have when seeing this is:

> *How do I find all the repositories (or entities) at once? I will be jumping between folders!*

To which one can answer: "*why would we want to find all repositories or entities?*" Humans interact with code for 2 reasons:

- to read it and understand it
- to change it

In my experience it is much more common to read code to understand a feature (for example "how is stock handled in products", "when is a product published", ...) rather than to understand how *all* repositories work or to re-read *all* the config files.

The same goes for writing or modifying code: we often work on fixing or improving a feature, rather that e.g. change all repositories at once. For example adding a new field on products is much easier to do when all the relevant files are in one place.

Dependencies and cohesion

When we think about it, what do `ProductRepository` and `BasketRepository` have in common? Both address different problems and should be completely decoupled, so why group them together?

On the contrary, `Product`, `ProductService` and `ProductRepository` are highly coupled (and for good reasons). These classes will interact together and change together over time. Let's recall the definition of cohesion:

> *Cohesion: degree to which the elements of a module belong together.*

"Agile design"

Another great consequence of that layout is that it leaves us more freedom when designing each module.

With the "classic" code structure it becomes a no-brainer to create a repository, map the entity using Doctrine, etc. That means duplicating and perpetuating existing solutions. One could argue that it's a good thing: it brings consistency

across the whole project. But it also kills creativity and does not encourage to ask the right questions, such as:

- Is Doctrine the best solution for this module? Do we even need an ORM?
- This repository contains only 2 small SQL queries, would it make more sense to get rid of it and bundle those in the service?
- Does creating an entity make sense in that particular case?

Just like Agile encourages team to find solutions that work *for them* (i.e. there are no one size fits all solutions), this approach to code structure encourages to think and design each module based on its use case.

The differences between a "Product" and a "Product" in e-commerce

When I joined Wizaplace I was very surprised by something in the code: the products (the basis for an e-commerce website) were modeled twice in two different ways:

- The product in the back-office can be modified by the seller, it can be in an incomplete state, without stock, etc.

  This product is editable (with all the validation rules that go with this) and it is stored in database using many tables.

- The product visible in the shop can be added to a basket and bought.

> This product is a projection of the back-office product, with all its data denormalized into a single table (no need for "joins" when retrieving it). It is read-only and only "completed" and validated products are visible in the shop. For those interested, this is a form of CQRS.

I was not impressed so much by the technical side of things, but rather by the fact that **this made a lot of sense business-wise**.

This "product read model" was not only a matter of caching or optimizing performances: we just do not do the same thing with those two concepts. Having two separate models in the code allows to separate the logic related to *catalog management* to the one related to the online shop.

What does it have to do with code organization?

With a "classical" code structure we would end up with 2 `Product` classes in the same folder, which would obviously be a problem. We could get around this by being creative with names but this is still confusing. Good naming does matter, but at another level. Here is an example of what we could have had:

```
src/
    Entity/
        Product.php
        ProductReadModel.php
    Service/
```

```
        ProductService.php
        ProductReadModelService.php
```

When we decided to reorganize the code into functional modules we were forced to better understand the domain and to better integrate its vocabulary and concepts into the code:

```
src/
    Catalog/
        Product.php
        CatalogService.php
    PIM/
        Product.php
        PIMService.php
```

Thanks to numerous exchanges with the product and domain experts, we managed to put words onto all of this:

- The **catalog**, like paper catalogs or the catalog of amazon.com, only contains validated and published products. Those cannot be edited, but they can be bought.
- The **PIM** (Product Information Management) is a kind of CRM for products: this is where categories, products, descriptions, prices and more can be managed. There are even software products dedicated to solving this problem.

The same concepts are being modeled into the code, but in different ways. For those familiar with Domain Driven Design you may be thinking of *Bounded Contexts* and you would be very right, however I do not think DDD should be a requirement to benefit from that kind of code structure.

To conclude, organizing code into domain modules is not a silver bullet but it forces to better understand the problem we are solving and better structure our code.

Comments

and ORM config files on the same level as business and front end files might give you some headache in the future...

Of course, you're not changing ORM nor front end frameworks each week, and you might not need some mocking or business/dao access for a new API so often...
Each specific project has its own context ! But don't lose the historic reasons why something became an habit :)

Reply • Share ›

Adam Matysiak • 2 years ago

We at HighSolutions are using this approach also and it's working pretty good ;). We like to have separate directories for models/repositories/events etc. to make structure clean.

Reply • Share ›

Attila Fulop • 2 years ago

We've been using this modular approach for a while and it works pretty well.
The headache usually comes when there are cross-cutting features, like creating invoices for orders (Invoice module vs. Order module).
In this case we either:
1. make a shameful choice and put this feature, say in the Invoice module, so that it'll have a dependency on the Order module, and go forward; or
2. make a "glue" module(s) that defines intermodular functionality.

I think both makes sense depending on the use-case and helps to better structure your application.

For Laravel we've also made a package called Concord that helps build Modules for Laravel Applications. This was built on top of Service Providers, so modules can be either in-app modules (mostly what's discussed in this article), or external (composer) packages that you can add to any application.

Reply • Share ›

mkarnicki ➚ Attila Fulop • 2 years ago

How does Concord compare to https://nwidart.com/laravel... ?

1 Reply • Share ›

Attila Fulop ➚ mkarnicki • 2 years ago • edited

Very similar :)
As far as I see Concord offers more for developing modules as external libraries (think as plugins).
One such feature is the ability to replace Eloquent models with an extended variant in the consuming application:
https://artkonekt.github.io...

^ | v • Reply • Share ›

mkarnicki ➚ Attila Fulop • 2 years ago

Thank you for your reply. Bookmarked, will certainly have a better look when choosing my tooling.

1 ^ | v • Reply • Share ›

Attila Fulop ➚ mkarnicki • 2 years ago

Will be tagged as 1.0 this week. In the upcoming versions I plan to simplify it bit and chop off some features that can be achieved with Laravel anyways. Any feedbacks are welcome on github ;)

1 ^ | v • Reply • Share ›

mkarnicki ➚ Attila Fulop • 2 years ago

Awesome, looking forward to it!

1 ^ | v • Reply • Share ›

Nicolaswidart ➚ Attila Fulop • 2 years ago

Modules are external libraries by default as they're just composer packages.
Abstracting Eloquent is also something that's possible of course since that's basic application design.

^ | v • Reply • Share ›

Przemysław Lib • 2 years ago

Good points.

More then few *Repository.php in a single folder and one loose orientation or is forced to memorize all of the names to switch between with alternative UI (eg GoToFile)

 • Reply • Share ›

Other projects: null, Bref, externals.io, Serverless PHP news, Serverless PHP training