William Durand

Fork me on GitHub

DDD with Symfony2: Making Things Clear

20 August 2013 — Clermont-Fd Area, France

Domain Driven Design also known as **DDD** is an approach to develop software for complex needs by connecting the implementation to an evolving model. It is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains.

It is possible to use this approach in a Symfony2 project, and that is what I am going to introduce in a series of blog posts. You will learn how to build an application that manages users through a REST API using **D**omain **D**riven **D**esign.

This is the second article of this series! You should read Folder Structure And Code First before digging into that one. This blog post is an attempt to fix a few misunderstandings coming from the first post. I recommend you to quickly jump to all links, they are valuable!

DDD Is Not RAD ¶

Domain Driven Design is **not** a **fast** way to build a software. It is not RAD at all! It implies a lot of boilerplate code, tons of classes, and so on. No, really. It is not the fastest way to write an application. No one ever said that DDD was simple or easy.

However, it is able to **ease your life when you have to deal with complex business expectations**. *How?* By considering your domain (also known as your business) as the heart of your application. Honestly, DDD should be used in a rather **large application**, with complex business rules and scenarios. Don't use it if you are building a blog, it does not make much sense (even if it is probably the best way to learn the hard way). So question is when to use DDD? I would say when your domain is very complex, or when the business requirements change fast.

There Is No Database ¶

In DDD, we **don't consider any databases**. DDD is all about the domain, not about the database, and Persistence Ignorance (PI) is a very important aspect of DDD.

With **Persistence Ignorance**, we try and eliminate all knowledge from our business objects of how, where, why or even if they will be stored somewhere. Persistence Ignorance means that the business logic itself doesn't know about persistence. In other words, **your Entities should not be tied to any persistence layer or framework**.

So, don't expect me to make choices because it works better with Doctrine, Propel or whatever. This is not how we should use DDD. We, as developers, need to become part of our business users domains, we need to stop thinking in technical terms and constructs, and need to immerse ourselves in the world our business users inhabit.

DDD And REST ¶

By now, I use a REST API as **Presentation Layer**. It is **perfectly doable** and, even if both concepts seem opposites, they play nice together. Remember that one of the **strengths** of DDD is the **separation of concerns** thanks to distinct layers. I am afraid that people think that it is not possible to play with both at the same time because nobody understands REST or HTTP.

Basically, you should not expose your Domain Model as-is over a public interface such as a REST API. That is why you should use Data Transfer Objects (DTOs). DTOs are **simple objects** that **should not contain any business logic** that would require testing by the way. A DTO could be seen as a PHP `array` actually.

What you should do here is to write a REST API that **exposes resources that make sense for the clients** (the clients that consume your API), and that are not always 1-1 with your Domain Model. See these resources as DTOs.

For instance, if you deal with *Orders* and *Payments*, you could **create** a *transaction* resource to perform the business operation `payOrder(Order $order, Payment $payment)` as proposed by Jonathan Bensaid in the comments of the previous article:

```
POST /transactions
orderId=123&paymentId=456&...
```

The **Application Layer** will receive the data from the **Presentation Layer** and call the **Domain Layer**. This `transaction` is a DTO. It is not part of the domain but it is useful to exchange information between the Presentation and the Application layers.

However, in the previous article I was able to directly map my `User` Entity to resources of type *users*. But if you look at the whole thing, I used a Serializer component to only expose some properties. That is actually another sort of DTO. The Entity is transformed to only expose data that are relevant for the clients (the clients that consume your API). So it is ok(-ish)!

Also, note that HTTP methods explicitly delineate commands and queries. That means **Command Query Responsibility Separation** CQRS **maps directly to HTTP**. Hurray!

So, What’s Next? ¶

In this series, I will introduce a more complex business, don’t worry! Hopefully I was clear enough to explain my choices regarding this series, and I fixed some misconceptions about DDD, RAD, and REST.

In the next blog post, I will introduce the **Presentation Layer**, new Value Objects such as the `Name` one, and more on **DDD**! At the end of the next post, you will basically get a CRUD-ish application. Yes, I know... CRUD is an anti-pattern, but it does not mean you should avoid it all the time. Creating new *users* make sense afterall.

The third post will allow you to create almost everything you need in the different DDD layers to build a strong and powerful domain, with complex logic, and so on. You may not get why DDD is great until that, so don't panic and stay tuned!

If you like this post or if you use one of the Open Source projects I maintain, say hello by email. There is also my Amazon Wish List. Thank you ♥

Related Posts

Reviewing the FlexiSpot Desktop Workstation 27 inches 13 Mar 2017

PhD: ✓ 16 May 2016

Patching Linux Kernel (Raspbian & CVE-2016-0728) 21 Jan 2016

My Life On The Internets: A Year Later 16 Jan 2016

Level Up 08 Sep 2015

[Video] Nobody Understands REST, but That's Ok ;-) 02 Jun 2015

On capifony and its Future 11 Apr 2015

Playing With a ESP8266 WiFi Module 17 Mar 2015

Comments

Comments for this thread are now closed ✕

Comments Community Ava

♡ Recommend 2 Share Sort by Best

Mathias Verraes • 5 years ago

I don't think creating users makes sense. The closest thing to creating users would be a mother giving birth to future users of your system :-)
You could RegisterAUser, or perhaps ImportAUser. And there's probably more detail involved if we start to really think about the domain: RememberAVisitor, TrackACustomer, GrantAdministrativePermissions, HireAnEmployee, EnrollAStudent, DesignateADriver, ... If you look at the real world, there's in fact very little that we "create". To rid ourselves of CRUD-think, we need to be not just persistence-agnostic, but software-agnostic.

5 ^ | v • Share ›

willdurand Mod → Mathias Verraes • 5 years ago

You are right. I should say "registering users". However, it results in a creation from the API perspective. That is a nice example to enlight the difference between the user entity and the resource.

1 ^ | v • Share ›

Mathias Verraes → willdurand • 5 years ago

It doesn't have to be. You're mapping entities to resources, and because HTTP has limited verbs, you map PUT to Create and Create to Register. Three words for a single concept in the Ubiquitous Language, that's a smell. So what to do?

What really like, is making RegisterAUser a Command, as in GoF. It's a representation of an intention of the user. In your model, it can be an object, and in your REST api it can be a resource. You can PUT a representation of a RegisterAUser command, and the backend decides what to do (eg handle it immediately, queue it, delegate it...)

... I need to find some time to blog about this stuff :-)

3 ^ | v • Share ›

willdurand Mod → Mathias Verraes • 5 years ago

Actually it is tricky with users, and it would work a bit better with Orders for instance. Anyway, I think I get your point now, I was on a rush earlier so I missed a part of your thoughts.

Assuming the domain expert says that registering a user will put him in a sort of (paper) registry. If we agree on the fact that registering users imply adding them to a collection (= this plain old paper registry), then we could deal with a collection of users in the REST layer. I think it is ok to map the POST verb to "register", there is no rule against that, and it is probably better than adding a verb as part of the resource name.
So, POST'ing to `/users` would register a user, and then we have one single word to describe the concept. That is what I wanted to say.

In the backend, the controller would use a RegisterUserCommand or a RegisterUserDTO, and voilà. Exposing a collection of "RegisterAUser" is not really REST compliant.

WDYT? Do you agree with the rest of the article?

1 ^ | ⌄ • Share ›

Mathias Verraes ➦ willdurand • 5 years ago

A command is not a verb, although it is expressed using a verb. A command is a message, eg a telegram that a general

^ | ⌄ • Share ›

Mathias Verraes ➦ Mathias Verraes • 5 years ago

...sends to the frontline. The

message is a resource, so it's perfectly fine to post a message command to a queue or collection of sorts. So a RegisterAUser resource doesn't conflict with Fielding at all.

• Share ›