

VIEW MODELS

Stop leaking your domain to your views

© 2017 David Redfern

View Models by David Redfern

E: info@somnambulist.tech

W: <https://github.com/dave-redfern/view-models>

© 2017 David Redfern

You may re-produce this talk provided the copyright notices remain intact and it is fully credited and linked to the GitHub repository.

```
class Post
{
    public function author(): PostAuthor
    {
        return $this->author;
    }

    public function title(): PostTitle
    {
        return $this->title;
    }

    public function content(): PostContent
    {
        return $this->content;
    }

    public function createdAt(): DateTimeImmutable
    {
        return $this->createdAt;
    }

    public function publishedAt(): ?DateTimeImmutable
    {
        return $this->publishedAt;
    }

    public function isPublished(): bool
    {
        return $this->publishedAt instanceof DateTimeImmutable;
    }

    public function isRecentlyPublished(): bool
    {
        return $this->isPublished() && $this->publishedAt->diff(new
DateTimeImmutable())->days < 10;
    }
}
```

RE-CAP...

- Better Entities
- Entities have business rules
- Entities control state
- Entities control validity
- How about removing getters?

See <https://github.com/dave-redfern/better-entities>

© 2017 David Redfern


Introduction

Before delving in, we should first review what it means to have Entities.

Entities are a core part of your domain, they represent real things that the business deals with; they have a unique identity that persists regardless of other changes e.g. a Person record may be identified by a Social Insurance Number. This number does not change, but the Person may move, gets older, may change name etc.

In the Better Entities talk we looked at how to better convey our domain terminology (Ubiquitous Language) and express that through better method naming, using Value Objects to control types and enforce validity at every step.

One of the items that was mentioned was removing all the getters...



REMOVE THE GETTERS?!

- K... what? For real?
- How do I read my entity?
- How do I get the state out?
- How do I display to users?

- **SHOULD** your output use your domain objects?

© 2017 David Redfern

Panic Ye Not

Remove the getters? Why would I want to do that?

Well consider: right now the entities actually have quite a large API that consists of reading state from the entity but also managing those transitions.

It's the part about managing transitions that is the problem:

Why are we using an object that can change state in a view to read the state? We are a single method away from doing "something bad"™.

More to the point why should our view be limited to what our Domain objects are concerned with. The more we share, the more we are tempted to place output related methods in our Entities and that is not what they care about. And then someone wants to create an API and the first step is to dump the entities as-is and now our API is tightly coupled to the domain model and that is almost certainly a very bad thing for the API.

```

class Post implements RaisesDomainEventsContract
{
    public static function create(PostAuthor $author, PostTitle
$title, PostContent $content)
    {
    }

    public static function createAndPublish(PostAuthor $author,
PostTitle $title, PostContent $content)
    {
    }

    public function publish(DateTimeImmutable $publishedAt = null)
    {
        $this->publishedAt = ($publishedAt ? : new
DateTimeImmutable());
        $this->updatedAt   = new DateTimeImmutable();
    }

    public function removeFromPublication()
    {
        $this->publishedAt = null;
        $this->updatedAt   = new DateTimeImmutable();
    }

    public function changeTitle(PostTitle $title)
    {
        $this->title       = $title;
        $this->updatedAt   = new DateTimeImmutable();
    }

    public function replaceContentWith(PostContent $content)
    {
        $this->content     = $content;
        $this->updatedAt   = new DateTimeImmutable();
    }

    public function leaveComment(Commenter $commenter, string
$comment)
    {
        $this->comments->add(new Comment($this, $commenter,
$comment));
        $this->updatedAt   = new DateTimeImmutable();
    }
}

```

BENEFITS

- Entities have a single role
- Cannot use them for views
- Can only change state
- Simpler API
- Remember Collections?
 - No problems now!

© 2017 David Redfern

Benefits from Removing Getters

If we remove the getters, it means our entities can ONLY change the state. It forces a single responsibility that is entirely focused on maintaining that state. Our APIs are much cleaner and the biggest win: we are not having to worry about exposing active Collection instances that could be modified beyond our entities control.

True: there will need to be concessions because updating / manipulating our entities may require Domain Services that will need some access, however we can deal with these on a case-by-case basis.

The other side-effect of no getters is that our entities are no longer appropriate to be used with views or exporting in API calls.

This is a good thing. It forces us to consider the information we are publishing and to be much more specific about what we choose to expose.

```
class EntityAccessor
{
    public static function get($object, $property)
    {
        $refObject = new \ReflectionObject($object);
        $refProp = $refObject->getProperty($property);
        $refProp->setAccessible(true);

        return $refProp->getValue($object);
    }
}

class PostTest extends TestCase
{
    public function testCreate()
    {
        $entity = Post::create(
            $pa = new PostAuthor('bob', new
EmailAddress('bob@e.ca')),
            $pt = new PostTitle('Test Post'),
            $pc = new PostContent('<p>This is a test post</p>')
        );

        $this->assertTrue(
            $pa->equals(
                EntityAccessor::get($entity, 'author')
            )
        );
    }
}
```

DOWNSIDE

- Testing is harder
- Need Reflection
- Still good to test assignment
 - Catches human typos
- Use VOs to check equality
- And my views?

© 2017 David Redfern

Downside

Testing becomes a lot more difficult because we cannot get at the internal data. However: this is nothing that some Reflection cannot resolve e.g. Symfony Property Accessor or a simpler EntityAccessor (see example code). You can have issues with private properties of distant parents, so the Symfony package is highly recommend for these cases.

Why check assignment?

Easy: copy/paste errors and typos! Did you fast duplicate (Cmd+D in PhpStorm) a bunch of assignments e.g. Collections and forget to update the property name? Checking the assigned value ensures you avoid this.

You could avoid that by generating protected/private setters and then using the setters to change the properties. It's an option but typos can still happen, even with code generation.

At the end of the day: how important is your data? If it is critical - add the tests and do data fuzzing to ensure they are working.



ENTER VIEW MODELS

a.k.a de-coupling your data representation from the domain



VIEW MODELS (VM)

- Provide data API for view(s)
- Read-only
- Encapsulate the VIEWS needs
- Disconnected from domain
- Can be complex
- Can be completely different to entities

© 2017 David Redfern

View Models (VM)

A view model is a plain-old PHP (PoPo) object that acts as a stand-in for our domain entities with the big difference being that it is customized for the views needs. It does not have to conform to the domains strict controls as it is disconnected from the domain persistent layer. We can make changes to it without fear of damaging our data.

The biggest part to remember is that VMs should be read-only. They do not need setters, or change methods. Instead they are focused purely on presentational needs. So if you need to “discover” state, you can place that logic in the VM. We can even add complex display logic to keep our templates lighter.

Ever had issues in your templates with branching if...elseif... logic? Use a VM to encapsulate it. Still need it? How about building view templates based on the state:


e.g.: instead of a single template that checks if a property equals something else, use that state to include/render a template with that combination of properties:

```
* {% if object.state == 'active' %}{% include 'partials/controls' %}{% endif %}
```

```
* vs
```

```
* {% include 'partials/' ~ object.state ~ '/controls' %}
```

No if/else needed. Lots of smaller views, but independent, no risks with changing logic - easier to test too.



VM CONS

- Does duplicate some logic
- Workflow “State” changes must be kept in-sync
- Temptation to share with entities (traits)
 - DON’T do it!
- Temptation for God objects
 - DON’T do it!
 - Be specific

© 2017 David Redfern

Cons / Gotchas

Because VMs tend to look a lot like Entities there's a strong temptation to encapsulate and “share” logic via traits or VOs or inheritance etc. You should avoid sharing - specifically traits and inheritance because a VM is read-only it should not have state changing nor should you adapt a trait because it needs view logic. If you keep traits namespaces separately for entities vs ViiewModels you can see in code if you have slipped up.

ValueObjects on the other hand you CAN share because they are immutable and encapsulate some domain property already - **however** - if you start adding helpers to your VOs - STOP! Remember the VOs primary purpose is for data-integrity in your domain not for presentation. You may wish to extend the VO instead and add helper methods there, ensuring you keep the domain VO for its own purpose. Using private properties in the base necessitates thinking very carefully and this can be a good thing.

Often it may end up being better to create a new VO instead just to avoid the problems with sharing a common base.

The biggest issue with creating VMs though: is the tendency towards making huge God objects that do everything everywhere. This is not what we want. VMs should be tailored to a particular usage. There ***might*** be commonalities, in which case these can be abstracted or pulled out into ViewModel traits (or a base class), but really if you are adding methods “because I need in this one situation something else” maybe you need a different view model.

Creating smaller, simpler and very specific VMs is preferably over monolithic models. Yes there is more to maintain, but if you change a views data, you want to be able to change it without breaking all your other pages and using separate VMs makes this trivial.

Again, it mostly depends on the size of your project and the range of representations you need.


```
class PostModel
{
    private $slug;
    private $title;
    private $author;
    private $content;
    private $comments;

    public function __construct($slug, $title, $authorName,
    $authorEmail, $content)
    {
        $this->slug      = $slug;
        $this->title      = $title;
        $this->author     = new PostAuthor($authorName, new
EmailAddress($authorEmail));
        $this->content    = new PostContent($content);
        $this->comments   = new ArrayCollection();
    }

    public function slug(): string
    {
        return $this->slug;
    }

    public function title(): string
    {
        return $this->title;
    }

    public function author(): PostAuthor
    {
        return $this->author;
    }

    public function content(): PostContent
    {
        return $this->content;
    }
}
```

EXAMPLE VM

- Only major Post details
- No ID (using slug)
- Re-using VOs
- Comments collection
- Getters (without get)

© 2017 David Redfern

Example PostModel ViewModel

Starting with the Post, we may create something like this example to begin with.

It quite closely follows the Post entity, except we are using the "slug" as the main identifier and not bothering with the id at all. Further: we are not using the PostTitle VO, instead these are the main properties of the PostModel, but the author and content is still going through our VOs because that seems useful.

Finally: we require all arguments during construction so our VM will be “valid” for the view. This includes a Comments collection that we can manipulate later on if needed (method not shown).

Following on from “Better Entities” we’ve dropped “get” from our methods, but we could have left it on - depends on what compatibility is needed with a template layer (if any).

Because we control the constructor and because this is a VM, we can instantiate the internal VO objects rather than requiring them as arguments. This provides some interesting usages (coming later).

```
class PostModel
{
    private $slug;
    private $title;
    private $author;
    private $content;
    private $comments;

    public function __construct($slug, $title, $authorName,
$authorEmail, $content)
    {
        $this->slug      = $slug;
        $this->title      = $title;
        $this->author     = new PostAuthor($authorName, new
EmailAddress($authorEmail));
        $this->content    = new PostContent($content);
        $this->comments   = new ArrayCollection();
    }

    public function slug(): string
    {
        return $this->slug;
    }

    public function title(): string
    {
        return $this->title;
    }

    public function author(): PostAuthor
    {
        return $this->author;
    }

    public function content(): PostContent
    {
        return $this->content;
    }
}
```

PROBLEMS WITH EXAMPLE

- Loads post content
 - Are we even using it?
 - Could be huge
- Do we want comments?
 - Only comment count?
- Published date?
- May need more than one VM

© 2017 David Redfern

Problems

This initial model requires the post content. What if it is never accessed? The content could be very large and if we are not going to use it in any capacity, better to not incur the cost of fetching it. We could make it nullable, but then we need logic to check “if null” or maybe we add a NullPostContent instead that returns empty strings when accessed.

By relying on the slug for identity, we’ve made a choice that slugs must be unique - did we enforce this in the rest of the application? Lets hope so!

We’ve added comments but do we always need the comments? Maybe we only want the comment count, or maybe this post does not need comments (or allow them at all).

We’ve left out any dates, but maybe this is not that important? More questions to ask that will depend on what our views actually need.

FETCHING THE VM

- Use a custom Repository
- Use PostRepository for QB
- DQL to fetch a “NEW” object
- Returns array of PostModels

- Coupled to entity structure
- Relies on Doctrine

```
namespace AppBundle\Services\ReadRepositories;

class PostModelRepository
{
    protected $posts;

    public function __construct(PostRepository $posts)
    {
        $this->posts = $posts;
    }

    public function findLatestPosts($limit = 10)
    {
        $qb = $this->posts->createQueryBuilder('p');
        $qb
            ->select(sprintf(
                'NEW %s(
                    p.slug, p.title, p.author.name, p.author.email,
                    p.content.content
                )',
                PostModel::class
            ))
            ->where('p.publishedAt <= :now')
            ->orderBy('p.publishedAt', 'DESC')
            ->setMaxResults($limit)
            ->setParameter(':now', new \DateTime('-14 days'))
        ;

        return $qb->getQuery()->getResult();
    }
}
```

© 2017 David Redfern

Ignoring the Problems, how do we load this?

In Doctrine we have multiple ways that we can load custom models. The first is using “NEW” syntax in a DQL query. This allows any type of object to be returned containing simple scalar data (note: we cannot select VOs in DQL, only their data). This can include aggregate numbers (e.g. comment count).

The fields must be selected in the order they will be loaded into the object. Previously we defined this as slug, title, author name/email and content so we must select the data (in DQL) in the same order.

Using sprintf() to set the model name, because it must be the Fully Qualified Class Name (FCQN) for the object and this is the easiest way to inject it.

The result of running this will be an array of PostModels containing our data.

Notes:

You can't select VOs as a whole thing e.g.: p.author - you have to specify the fields from the VO: p.author.name, p.author.email.email.

Dates will be hydrated using the datetime/date mappings so you may get back DateTime instances - be prepared.

You are still actually coupled to your domain object model because DQL queries your entities, not the datastore directly.

This is not possible using an ActiveRecord ORM. Closest is returning a simple array or stdClass (in PHP).

NO DQL JUST PDO

- Can fetch into a Class
- No DQL needed - pure SQL
- Returns an array of PostModels
- Watch out!
 - Constructor calling buggy
 - Selected fields become properties (naming)
 - PROPS_LATE sets props after constructor

```
class PostModelPdoRepository
{
    protected $conn;

    public function __construct(Connection $conn)
    {
        $this->conn = $conn;
    }

    public function findLatestPosts($limit = 10)
    {
        $query = '
            SELECT p.title_slug, p.title_title, p.author_name,
                   p.author_email, p.content_content
            FROM posts p
            WHERE p.published_at <= :now
            ORDER BY p.published_at DESC
        ';

        $stmt = $this->conn->prepare($query);

        $stmt->setFetchMode(PDO::FETCH_CLASS, PostModel::class);

        $stmt->execute([
            ':now' => (new \DateTime('-14 days'))
                    ->format('Y-m-d H:i:s')
        ]);

        return $stmt->fetchAll();
    }
}
```

© 2017 David Redfern

What about without Doctrine or DQL?

If you don't like the look of DQL or using Doctrine, you can accomplish the same thing (more or less) using straight PDO!

Upshot is that the query becomes pure SQL that could be created using a QueryBuilder and then we use PDOs fetching modes to create objects for us.

Danger Will Robinson Danger!

Down-side here is that PDO will associate the properties FIRST by default and THEN trigger the constructor. So if your constructor defines "nulls" and assigns them, your objects will be empty - or worse error a lot. We can defer this by using FETCH_PROPS_LATE - however this means your constructor is fired with no data.

As the selected fields will become properties, they should have the same name as the properties you defined otherwise they will be directly created. This means that your fields must follow PHP property naming limitations.

Basically: you need to build your VMs for either PDO fetching OR DQL fetching. Obviously if you are using Doctrine DQL is the easy option, PDO is universal - most frameworks have a PDO adaptor. The choice though is yours.

Note: we would actually go the route of a custom Connection object and not use Doctrine at all. We don't want unnecessary dependencies on the domain model. Though to get the best results we would want some custom hydration logic too.

```

class PostModel
{
    public function comments(): ArrayCollection
    {
        return $this->comments;
    }

    public function attachComments(ArrayCollection $comments)
    {
        $this->comments = $comments;
    }
}

class PostModelRepository
{
    protected function loadPostWithComments(PostModel $post)
    {
        $qb = $this->posts->createQueryBuilder('p');
        $qb
            ->select(sprintf(
                NEW %s(c.commenter.name, c.commenter.email, c.comments)',
                CommentModel::class
            ))
            ->innerJoin('p.comments', 'c')
            ->where('p.title.slug = :slug')
            ->orderBy('c.createdAt', 'ASC')
            ->setParameter(':slug', $post->slug());
        ;

        $post->attachComments(
            new ArrayCollection($qb->getQuery()->getResult())
        );

        return $post;
    }
}

```

HOW TO GET THE COMMENTS?

- Separate method
 - Load comments as needed
 - Load from slug on Post
 - Add method to set comments
-
- Limit comments?
 - Paginate?
 - Easy to do!

© 2017 David Redfern

Loading Comments for a Post

Since the VMs are disconnected from the domain (and Doctrine in this case) there is no automatic loading of relationships. Instead we have to handle this ourselves. Fortunately this is another easy step to make.

First we add a method to allow setting the comments in our PostModel (or we add a PostWithComments model), we could use a more advanced Collection object or a simple array instead. Here we are using the default ArrayCollection from Doctrine because it has the same API as that of the Post Entity.

Next: because we are treating comments as being contained inside the Post, we don't expose a comment Repository directly. Instead we add a loading method to the PostModelRepository that can load comments from a Posts slug. In this example we are using DQL to fetch the comments and associating them to the PostModel. The method is protected to prevent it from being used externally, however if we wanted to allow pagination, or searching etc within comments we could either:

- * split out a new CommentRepository and use a QueryBuilder from the PostRepository
- * make the method public and expand the interface

We can even add support for bulk loading data by changing the method to allow a Collection of PostModels and load all comments by slug, though this may not be very performant if there are hundreds of comments per post.

TEST THE RESULTS

- Add functional tests
- Persists via Entities
- Fetch as PostModels
- VOs were instantiated
- Comments loaded + VOs

See *ViewModelIntegrationTest.php* in sample code

© 2017 David Redfern

Functional Testing

The slide features a partial dump of the PostModel from a PHPUnit run using symfony/var-dumper.

Of course we must test that our approach is working and the best way to do this is via a functional test. Depending on your setup this could be relatively straight forward or... somewhere between a nightmare and impossible. For the simple example source code it is possible to create an in-memory SQLite DB store and run tests against that. You may require a lot more setup; either a dedicated testing database or careful usage of transactions to prevent data from being persisted.

The benefit of an in-memory SQLite DB is speed. It is extremely fast to setup and teardown and there's nothing left over.

A good thing with a full end-to-end functional test is that the whole process of creating data, changing its state, persisting it and reconstituting it can all be done in isolation and we can be safe in the knowledge that everything is working as it should.



GOING FURTHER

Custom helper methods



```

class PostModel
{
    public function authorLink()
    {
        return sprintf('<a href="mailto:%s?subject=Re:%s">%s</a>',
            $this->author->email(),
            urlencode($this->title),
            $this->author->name()
        );
    }

    public function postLink($route)
    {
        return sprintf('<a href="%s/%s">%s</a>',
            $route, $this->slug, $this->title
        );
    }

    public function reverseCommentOrder()
    {
        return new ArrayCollection(
            array_reverse($this->comments->toArray())
        );
    }

    public function findCommentsBy(Commenter $commenter)
    {
        return $this->comments->filter(
            function ($comment) use ($commenter) {
                /** @var CommentModel $comment */
                return $comment->commenter()->equals($commenter);
            }
        );
    }

    public function findCommentsContaining(string $keyword)
    {
        return $this->comments->filter(
            function ($comment) use ($keyword) {
                /** @var CommentModel $comment */
                return $comment->contains($keyword);
            }
        );
    }
}

```

MODEL HELPERS

- Embed linking methods
 - Link to
 - Call to
 - Contact Author etc.
- Icons on state
- Collection helpers

© 2017 David Redfern


Helper Methods

Now we have our ViewModels being returned we can inject them into our views and start figuring out what helper methods we need. These will be discovered as you use the models and render titles, links etc. Generally if you start to implement if/else constructs in your views, chances are you can move this down into the VM and remove the logic from your template.

Now, this does not always apply. Your VMs should not become tied to the templating system directly and realistically they should not only output HTML (what about API responses for example). However: depending on your needs you may decide to implement some renderer methods like the examples above. There is a temptation to move the *authorLink* into the PostAuthor VO, however this would be a mistake - instead create a new ViewModel VO for the PostAuthor that includes it, or implement a ViewHelper in your templating system to handle rendering the VO instead.

In Twig, you can create custom functions that can implement the rendering logic instead and keep the models clean of actual output. Other template systems should have similar functions (or use a specific include). The end result should be a consistent UI.

Of course there is nothing to stop you implementing custom VMs for your API vs your HTML views and from their additional custom repositories to fetch the data. It all depends on your needs and how much you wish to isolate yourself from changes. Of course the more VMs you have the harder it becomes if you radically change your data structures, or if your workflow rules change.



A MODEL FOR ALL OCCASIONS

- Reporting
 - Use custom models
 - All data is read-only
 - Keep domain out
 - Maintain context
 - Use report language
- Do you even need a model?

© 2017 David Redfern

More Examples

Perhaps the best use-case for completely segregated data is reporting. Reports by their nature are read-only, but usually have nothing to do with the domain or views. They may have their own terminology (Bounded Context), with highly specialized needs. A common mistake with reports is to “just use the entities / repos” and you end up with huge repositories and methods that exist purely for reports.

Stop.

Reports are a separate thing, build them separately. In fact many reports are so different from one another they require handling in a completely separate manner. This is where utilizing your data storage engine really comes into its own. If you are using an RDBS like Postgres or MySQL - then use it! Generate report tables or a sync'd reporting database to run reports from and use every trick in the book to get performant reports.

Often with reports, do you even need to use models? Or is the output good enough as a CSV file (XLSX etc)? In that case you might not need to use a View Model at all.



CQRS

- Design pattern
- Separate changing / reading
 - Commands change
 - Queries read
- Commands use entities
- Queries return View Models

- Related: Event Sourcing

© 2017 David Redfern

Ultimate Goal: CQRS

The end result of these changes is a complete separation of how we change data to how we read it. This is exemplified in the design pattern: Command Query Responsibility Separation (Segregation) or CQRS.

Under CQRS Commands are issued that manipulate the domain state. The state changes raise domain events that can trigger updates to reports, other systems to respond etc. These changes are then reflected in the Queries and hence the data people see. Commands are typically dispatched to Command Handlers and it is these that load the Entities, manipulate them and then persist changes. The Query logic is completely separate from the Command logic, with different repositories.

On the other side, Queries are used to read data - but it is always read-only. Queries never change the data. Only Commands can change data.

This approach requires a lot of thought and planning, but provides the most scalable infra-structure because if you need a new View, simply add the specific Queries and handling logic, completely independently of the main system.

In our examples we would not re-use the PostRepository in our PostModelRepository. It would have its own dedicated connection and custom hydration logic. Re-using the EntityRepository is a fast but limiting move. Fortunately it would be easy enough to factor out through dependency injection and since we have unit tests, we can safely change the internals without affecting the output.

Related to CQRS is Event Sourcing. This is the ultimate expression of CQRS where the data store is actually the events that have been generated by changing the entities. The event stream as it is called, can then be replayed to restore the object state. It provides complete audit logs and point in time debugging because you can reply the events to a specific point. Event Sourcing requires a completely decoupled Query layer, because the event store is just events, usually in the form of JSON objects or similar unstructured data. The Queries are actually run against “projections” that are built from the event stores (or updated by the domain events). This system is extremely scalable but difficult to implement and not everything needs to event sourced.



MORE...

- Command Query Responsibility Separation
- Event Sourcing
- Revisit "Better Entities"
 - <https://github.com/dave-redfern/better-entities>
- Example code available
 - <https://github.com/dave-redfern/view-models>

© 2017 David Redfern

Example Code

A complete example of the Post and Comment is available at: <https://github.com/dave-redfern/view-models>
It includes a selection of unit tests including integration / persistence tests with Doctrine.