

# BETTER ENTITIES

---

*Getting more from your Domain Objects*

*a.k.a how to create Doctrine entities that don't suck!*

© 2017 David Redfern


Better Entities by David Redfern

E: [dave@scorpioframework.com](mailto:dave@scorpioframework.com)

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

© 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.



## RE-CAP...

- Presenter notes are on the slides
- What is an Entity?
- Why use entities?
- What is the role of an Entity?
- DDD FTW!

© 2017 David Redfern

### What is an Entity?

An entity is an object that has a thread of continuous identity throughout the domain life-cycle, that is not derived from its properties i.e. there is an external ID (UUID, sequence number etc). The term comes from Domain Driven Development. Example: Country object, identity could be the ISO number. For example: a Person cannot be uniquely identified by its name property alone; it needs another form of identity to distinguish it. In addition if a Person changes name, that is not a new person, but the same individual.

Very important to remember that an Entity has nothing to do with persistence! The entity is ignorant of how it will be persisted (or if it even is persisted). You should design entities that reflect your problem domain NOT how you want to persist them.

### Why use Entities?

Entities encapsulate an aspect of the domain; a particular actor that plays a crucial role. This is expressed through the properties and methods that are exposed. These methods should reflect the language of the domain and ensure that the entity is never allowed to be in an invalid state. The entity, in essence, allows us to express in code the processes of the domain.

### What is the role of an Entity?

The Entity is responsible for its own state and validity. It may be part of a larger group of entities (Aggregate), but individually an entity is responsible for its self and its own changes in state. All entities should be self-validating and only expose methods that allow the entity to move from one valid state to another.

### For more detail on DDD look into:

*Eric Evans* - Domain Driven Design, Tackling Complexity in the Heart of Software (Blue Book)

*Vernon Vaughn* - Implementing Domain Driven Design (Red Book)

```
/**
 * @ORM\Entity
 */
class Post
{
    const NUM_ITEMS = 10;

    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $title;

    public function __construct()
    {
        $this->publishedAt = new \DateTime();
        $this->comments     = new ArrayCollection();
    }

    public function setTitle($title)
    {
        $this->title = $title;

        return $this;
    }

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

    public function setSlug($slug)
    {
        $this->slug = $slug;

        return $this;
    }
}
```

# TYPICAL ENTITY

- Protected or private properties
- Getters and setters
- Annotations
- Mostly empty constructor
- Life-cycle methods
- Many Symfony and Doctrine examples

© 2017 David Redfern

## Full Entity Code

We are going to use the example from the Symfony docs as our starting point with a few enhancements (get/set methods, some assertions).

```
<?php // http://symfony.com/doc/current/best_practices/business-logic.html#doctrine-mapping-information
namespace AppBundle\Entity;
```

```
use Doctrine\ORM\Mapping as ORM;
use Doctrine\Common\Collections\ArrayCollection;
use Symfony\Component\Validator\Constraints AS Assert;
```

```
/**
 * @ORM\Entity
 */
class Post
{
    const NUM_ITEMS = 10;

    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     * @Assert\NotEmpty
     */
    private $title;

    /**
     * @ORM\Column(type="string")
     * @Assert\NotEmpty
     */
    private $slug;

    /**
     * @ORM\Column(type="text")
     * @Assert\NotEmpty
     */
    private $content;
```

```
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Doctrine\Common\Collections\ArrayCollection;

/**
 * @ORM\Entity
 */
class Post
{
    const NUM_ITEMS = 10;

    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $title;

    /**
     * @ORM\Column(type="string")
     */
    private $slug;

    /**
     * @ORM\Column(type="text")
     */
    private $content;

    /**
     * @ORM\Column(type="string")
     */
    private $authorEmail;

    /**
     * @ORM\Column(type="datetime")
     */
    private $publishedAt;
```

# PROBLEMS...

- Purpose
- Validity
- Mutability
- Annotations
- Persistence knowledge
- We can do better!

© 2017 David Redfern

## Purpose

The entity has no clear purpose. It is what Martin Fowler refers to as “anemic”. It’s basically nothing more than a Data Transfer Object and conveys nothing about our domain.

## Validity

If the entity is used outside of Symfony (in this example) the validation will not run. How can we enforce validity with framework dependencies? Validity is a core concept for the entity, deferring it to a secondary system through non-code comments (that could be stripped) defeats the purpose of using an entity.

## Mutability

All the properties (except ID) can be modified at any time in any order. What does it mean to set these? Do some need setting together? How do we know?

## Annotations

The annotations are adding nothing to the actual entity. The persistence instructions, confuse the role of the entity with storing it - they do not belong here. Assertions as already pointed out cause problems. Basically: if you strip the comments, can this still work? Answer: no!

## Persistence Knowledge

Related to annotations, an entity should not care about persistence, but by using annotations we are conforming to external limitations. Worry about persistence later.

## TYPICAL AR MODEL

---

- No clear API
- Lots of “magic”
- Object reflects CURRENT table design
- Attributes can be overloaded
- Difficult to use Domain language
- Difficult to ensure state

```
namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];
}
```

© 2017 David Redfern

### ActiveRecord Models

This example is from: <https://github.com/laravel/laravel/blob/master/app/User.php> of Eloquent.

Ignoring the tight coupling to the data-source, AR discourages domain language and well structured objects.

In particular with Eloquent models: extra functionality is regularly bolted on via traits (see Notifiable for example). It is very hard to maintain any form of state, or to prevent changes outside of the object because of magic method calls and implementation of \_\_set,call,get and callStatic. Worse, many calls result in QueryBuilder instances.

Yet more problems are encountered when you consider that the contents of the “model” reflect the current state of the table that the model reflects. You cannot isolate from changes to DB schema because the properties are directly accessed. This can be guarded through the use of mutators, but its is very messy.

A mutator in Eloquent is custom get method that can change the value or derive a value when it is accessed. It is part of the “magic” provided by the \_\_get implementation. See Eloquent for more information: <https://laravel.com/docs/5.4/eloquent-mutators>

Biggest problem is the lack of a coherent, domain centric API - the main API is about querying persisted data (i.e. the database) - NOT the domain.



**LETS MAKE IT BETTER!**

---

*And make a bunch of contentious statements...*



```
class Post
{
    const NUM_ITEMS = 10;

    /**
     * @var int
     */
    private $id;

    /**
     * @var string
     */
    private $title;

    /**
     * @var string
     */
    private $slug;

    /**
     * @var string
     */
    private $content;

    /**
     * @var string
     */
    private $authorEmail;

    /**
     * @var DateTime
     */
    private $createdAt;

    /**
     * @var DateTime
     */
    private $publishedAt;

    /**
     * @var ArrayCollection|Comment[]
     */
    private $comments;
}
```

# ANNOTATIONS

- Bad:
  - Annotations are not code!
  - Tightly couple framework
  - Decreases readability
  - Another config language to learn
- Instead:
  - Use config files
  - Free annotations
  - Easier to convey domain

© 2017 David Redfern

## Free Annotations

Removing annotation code (assert, ORM and all the others) frees the annotations to be used for documentation only. After all, in PHP annotation are not code! This avoids issues with using e.g. Doxygen for documentation.

Of course the principal benefit / reason is: our code always running and being explicit with what will happen, with no external dependencies / setup required.

Many people will say that annotations are convenient, but consider this: if you delete all the annotations (every docblock comment), will your code still run? Will it still validate? If not then you need to rethink what you are doing because a junior may accidentally delete an annotation statement or modify it incorrectly, causing hard to track down bugs or fatal errors that may not have been picked up.



```
class Post
{
    const NUM_ITEMS = 10;

    /**
     * @var int
     */
    private $id;

    /**
     * @var string
     */
    private $title;

    /**
     * @var string
     */
    private $slug;

    /**
     * @var string
     */
    private $content;

    /**
     * @var string
     */
    private $authorEmail;

    /**
     * @var DateTime
     */
    private $createdAt;

    /**
     * @var DateTime
     */
    private $publishedAt;

    /**
     * @var ArrayCollection|Comment[]
     */
    private $comments;
```

## SETTERS

- Remove ALL the setters
  - Need to control state\*
  - No arbitrary changes
  - No partial changes
  - Cleaner interface

*\* state changes are coming up!*  
© 2017 David Redfern

### Removing Setters

Setters allow any process to change the entities state at any time in any order, in an invalid way. We really can't allow that. The entire point of the entity is to maintain its state so we do not define setters. We will enforce state changes instead. By “state change” we mean moving the entity from one representation to another. This change should be transactional i.e.: it either wholly occurs or does not - it must leave the entity in a valid state. Examples are coming later.

By removing the setters we vastly simplify the entity API and prevent invalid changes being made, otherwise: how would we enforce linked properties?



# INSTANTIATION

---

- Define required properties
- Enforce factory methods
- Name methods after domain terms
- Decide on identity scheme
  - Do you need identity now?

```
class Post
{
    . . .

    private function __construct($title, $slug, $content)
    {
        $this->title    = $title;
        $this->slug     = $slug;
        $this->content  = $content;
    }

    public static function create($title, $slug, $content)
    {
        $entity = new static($title, $slug, $content);

        return $entity;
    }

    public static function createAndPublish($title, $slug, $content)
    {
        $entity = static::create($title, $slug, $content);
        $entity->publish();

        return $entity;
    }
}
```

© 2017 David Redfern

## Creating Instances

Creating entity instances can be an interesting challenge. Do we allow them to be directly instantiated or not? What about factory methods? Should they go in the entity or a factory class?

In keeping with programming in general, the answer is: “it depends”.

If the entity is particularly complex to create and has external dependencies (identity generation, data services etc), then a factory class is probably better. However: for relatively straight forward entities, static create methods are perfectly fine, in fact they should be encouraged! It is after all a responsibility of the entity in question.

## Identity Scheme

An important consideration with your entity is how it derives its identity. Ask this question: do you need to know its identity ahead of persistence?

If you do not require an immediate reference you can rely on deferred identity from another source. If you require the identity ahead of time, then generate it and make the identity a requirement of constructing the object.

In the example of the post, is the slug the identity or a surrogate identity? Should the slug change with the title?

# ENFORCE STATE CHANGES

---

- Define explicit methods
- Require ALL arguments
  - Enforce types
- Make methods statements
  - no return value
  - use the domain language

```
class Post
{
    . . .

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

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

    public function changeTitleAndSlug(string $title, string $slug)
    {
        $this->title = $title;
        $this->slug = $slug;
    }

    public function replaceContentWith(string $content)
    {
        $this->content = $content;
    }
}
```

© 2017 David Redfern

## Enforcing State Changes

A big part of the entity is to enforce and encapsulate how it changes state. This should be carefully controlled so that only valid state changes are actually allowed to be performed. When the entity's state is going to change, we should use the domain language to reflect this change through specific method calls. Generally methods will following action type names e.g.: “do this thing”, following the principle of tell don't ask.

In this example: publish could be undone by unpublish or clear / hide / conceal / removeFromPublication - whatever we decide to call this process should be the name used here.

Updating the actual post content, are these independent actions? Should the post title / slug be changed outside of the content? If so these are separate actions that should be clearly defined. However it is decided the method names should reflect our domain language.

# STRING CONSTANTS

---

- Use Enumerations
- Type hint class type
- Enumeration is already valid
- Works with refactoring tools
- Use domain language

```
class UserAddress
{
    protected $type;

    public function __construct(Address $address, AddressType $type)
    {
        $this->address = $address;
        $this->type     = $type;
    }
}

/**
 * Class AddressType
 *
 * @method static AddressType HOME_ADDRESS()
 * @method static AddressType WORK_ADDRESS()
 */
final class AddressType extends AbstractEnumeration
{
    const HOME_ADDRESS = 'home';
    const WORK_ADDRESS = 'work';
}
```

© 2017 David Redfern

## Replace String Constants

Strings that are set through constants can be replaced with Enumerations: <https://github.com/eloquent/enumeration>

*from the docs for Eloquent Enumeration*

In terms of software development, an enumeration (or "enumerated type") is essentially a fixed set of values. These values are called "members" or "elements".

An enumeration is used in circumstances where it is desirable to allow an argument to be only one of a particular set of values, and where anything else is considered invalid.

We can now type hint specifically these values, removing any need to check for specific type strings in the entity. The enumeration is similar to a ValueObject (see later), except it is the value itself.

When dealing with Doctrine, there is no native support for enumerations so custom Types would need to be created to have automatic conversion to/from the persistence store. Alternative, the “get” method could convert a non-object value to the appropriate object through e.g.: `AddressType::memberByValue($this->type);`

```

class Post
{
    private function __construct(string $title, string $slug, string $content)
    {
        Assert::lazy()->tryAll()
            ->that($title, 'title')->notEmpty()->maxLength(100)
            ->that($slug, 'slug')->notEmpty()->maxLength(64)
            ->that($content, 'content')
                ->notEmpty()->minLength(100)->maxLength(65000)
            ->verifyNow()
        ;

        $this->title = $title;
        $this->slug = $slug;
        $this->content = $content;
    }

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

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

    public function changeTitleAndSlug(string $title, string $slug)
    {
        Assert::lazy()->tryAll()
            ->that($title, 'title')->notEmpty()->maxLength(100)
            ->that($slug, 'slug')->notEmpty()->maxLength(64)
            ->verifyNow()
        ;

        $this->title = $title;
        $this->slug = $slug;
    }
}

```

# VALIDATION

- Core concept of the entity
- Always check data
- Use type hints
- Use scalar type hints
- Use Exceptions
- Change assertions when needed
- Avoid framework dependencies
- Entity validity != Application validity

© 2017 David Redfern

## Validating and Asserting

We always want to enforce validity in our entity, that is its role after all. This should be done through Assertions - simple validations that are easily understood and built into the entity itself. You can role your own Assertion library, however Benjamin Eberleis Assertion library works very well, and is very light weight - perfect for entity validation.

It is important to note that Entity validation is not the same thing as Application validation. An entity cannot enforce things like “unique slugs” - entities have no persistence knowledge. However a title cannot be more than 100 characters and must be ASCII - that is a property of the entity and should be enforced.

Something to note with assertions is that constructor assertions are not necessarily the same as ones used when the entity state changes. Consider: during entity instantiation there is an optional (null) argument. It is not required when creating the entity, but when the state is changed that argument now **IS** required and may have minimum requirements. Avoid the temptation to use a general set of rules for the entity - you will end up with complex, branching if statements.

See: <https://github.com/beberlei/assert>

# APPLICATION VALIDATION

- Use framework validators
- Validate to application specs
- Enforce access checks
- Enforce uniqueness
- Enforce complex rules
- Transform data for the domain

```
class EnquiryFormRequest extends AppRequest
{
    /**
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * @return array
     */
    public function rules()
    {
        return [
            'email' => 'email|max:255|required_without_all:phone_number',
            'phone_number' => 'numeric|required_without_all:email',
            'name' => 'nullable|max:255',
            'notes' => 'nullable|max:10000',
        ];
    }

    /**
     * @return array
     */
    public function messages()
    {
        return [
            'email.required_without_all' => 'Required if no phone',
            'phone_number.required_without_all' => 'Required if no email',
        ];
    }
}
```

© 2017 David Redfern

## Application Validation

Application level validation supports our domain. It is used to apply all the validations that are beyond the scope of the domain itself. As this layer is likely using some for of framework, we should fully utilise the features of that layer. e.g. Laravel Validator, Form Requests; Symfony Forms / Validators etc.

In Laravel we would use FormRequests to validate form data, before passing it into our domain objects.

For example: validating phone numbers is very complex. In our domain entity we would enforce the requirement of E164 format numbers (full international format with country code so +1519...), but in our form we could accept numbers in many different formats:

- \* 519 123-4567
- \* 15191234567
- \* (519) 123 4567
- \* 1 (519) 123-4567...

The form request / transform can validate and return an international number: +15191234567 and our entity will validate that.

Other examples are:

- \* username uniqueness
- \* password strength (the entity gets the hashed password)
- \* specific domains on an email e.g. must be a company email address etc.

Once our inbound data has been validated and transformed we then pass it into our domain. At that point we should instead of using a HTTP request, switch to a Domain Input object that disconnects the request context from the domain. The data can then be mapped to the domain objects. This could be via a CommandBus, an input mapper or whatever else we need. See: <https://github.com/dave-redfern/laravel-domain-input-mapper> as an example.



# VALUE OBJECTS

---

*Better separation of responsibilities*





## VALUE OBJECTS (VO)

- Object has identity through its properties
- Compare based on properties
- Immutable
- Move validation to VO
- Add domain logic to VO
- Group related properties

```
class EmailAddress
{
    private $value;

    public function __construct($value)
    {
        Assert::that($value, null, 'email')
            ->email()->notEmpty()->maxLength(100);

        $this->value = $value;
    }

    public function __set($name, $value)
    {
        // don't allow setting anything
    }

    public function __toString()
    {
        return $this->toString();
    }

    public function toString()
    {
        return (string)$this->value;
    }

    public function equals($test)
    {
        if (__CLASS__ === get_class($test)) {
            return ((string)$test === (string)$this);
        }

        return false;
    }
}
```

© 2017 David Redfern

### Value Objects

So far we've only considered the basic structure of our entity; but we can go further! Value Objects allow related properties to be grouped together; if one changes the others probably have to change as well. If we group them into a single unit, we can move our validation logic to the VO and it is now reusable; better yet we can safely type hint the value object type and know that only a valid VO can be passed to our entity.

Value objects are basically entities that have no thread of identity: they are compared on their properties so two VOs are equal if they are the same type and have the same property values e.g.: a Person may have a Name VO that compares the type (name) and the internal properties (name, nickname, title).

There are a couple of ValueObject libraries for common types, but again often these actually are more hassle than they should be because they won't fit cleanly in your domain. Always favour your domain and its language.

To make VOs a little easier you could create a base class that implements naive comparison and a magic `__set()` to prevent additional properties being added. Just remember that VOs must be immutable to get the benefits.

### Why Immutability?

Any time we introduce the ability to modify our entity / domain state we introduce the risk of allowing invalid state to be entered. In the case of value objects, they represent an actual value that an entity has. If this value changes, then it needs changing to a new value i.e. a new instance of that type. For example a person has a "weight". If their weight changes, it is not the measure of the weight that changes but the property belonging to that person. Further: imagine if we shared the VO value for 80kgs with other entities and then changed it. All entities referencing that weight would be updated which would be wrong.

Good examples:

<http://wiki.c2.com/?ValueObjectsShouldBeImmutable>

<https://stackoverflow.com/questions/4581579/value-objects-in-ddd-why-immutable>



# VO DANGERS!

```
class EmailAddress
{
    private $value;

    public function __construct($value)
    {
        Assert::that($value, null, 'email')
            ->email()->notEmpty()->maxLength(100);

        $this->value = $value;
    }

    public function __set($name, $value)
    {
        // don't allow setting anything
    }

    public function __toString()
    {
        return $this->toString();
    }

    public function toString()
    {
        return (string)$this->value;
    }

    public function equals($test)
    {
        if (__CLASS__ === get_class($test)) {
            return ((string)$test === (string)$this);
        }

        return false;
    }
}
```

© 2017 David Redfern

- VOs are part of your domain
  - Use domain language
- VOs should be immutable
  - Don't pass in entities
  - Don't change state
- Don't share between domains
- Don't share between projects
  - Username is not always the same thing
  - Validations can change

## Value Objects Gotchas

Value objects are a core component of your domain, they should represent that therefore you should not share ValueObjects across domains and especially not across projects. The requirements in one domain for e.g. an EmailAddress may not be the same in another. Username is another example; what does it mean to have a “valid” username in your domain?

If you start sharing VOs you will lose your language - and that is not the goal. So yes, it will mean duplicating code, sometimes within the same project but if this ensures that you keep your language, this will help in the long run.

Alternatively consider this: you use a generic “authentication” VO that uses “identifier, password”. What is identifier in your domain? It has no meaning, it conveys nothing. Further: how is it validated? If your domain has Users identified by an EmailAddress, this VO does not convey that nor would it validate an EmailAddress. You could create a generic VO that takes an Authenticable VO that implements an interface, but what does this get us? It does not help our domain.

Then again it could: it all depends on how your domain is structured.

```
class PostAuthor
{
    private $name;
    private $email;

    public function __construct(string $name, EmailAddress $email)
    {
        Assert::that($name, null, 'name')
            ->notEmpty()->maxLength(100);

        $this->name = $name;
        $this->email = $email;
    }

    public function __toString()
    {
        return $this->toString();
    }

    public function toString()
    {
        return (string)$this->name;
    }

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

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

    public function equals($test)
    {
        if (__CLASS__ === get_class($test)) {
            return (
                $this->name === $test->name() &&
                $this->email->equals($test->email())
            );
        }

        return false;
    }
}
```

## POST AUTHOR

---

- Replace AuthorEmail with VO
- VO reflects domain language
- VO can limit data access
- Enforce validity
  - Consistent validation
  - Simplifies entity

© 2017 David Redfern

### Replace the AuthorEmail property

AuthorEmail doesn't tell us much about who authored the post, nor really convey the actual domain language we use. Instead we would discuss who owns the post and what information we actually need. In this case, it was decided there would be a PostAuthor comprising a name and EmailAddress. This can be encapsulated in a value object that enforces these two properties.

In your domain this may include another identifier if the PostAuthor is actually linked to a User. There's another discussion here about Bounded Contexts, but that is for another time.

```
class Post
{
    private function __construct(PostAuthor $author, string $title,
string $slug, string $content)
    {
        Assert::lazy()->tryAll()
            ->that($title, 'title')->notEmpty()->maxLength(100)
            ->that($slug, 'slug')->notEmpty()->maxLength(64)
            ->that($content, 'content')
                ->notEmpty()->minLength(100)->maxLength(65000)
            ->verifyNow()
        ;

        $this->title = $title;
        $this->slug = $slug;
        $this->content = $content;
    }

    public static function create(PostAuthor $author, string $title,
string $slug, string $content)
    {
        $entity = new static($author, $title, $slug, $content);

        return $entity;
    }

    public static function createAndPublish(PostAuthor $author,
string $title, string $slug, string $content)
    {
        $entity = new static($author, $title, $slug, $content);
        $entity->publish();

        return $entity;
    }

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

## USING POST AUTHOR

---

- Author now required
  - Enforce in methods
- Conveys domain language
- Change accessor name
  - post->author->name
  - post->author->email

© 2017 David Redfern

### Using the PostAuthor VO

Now we have a ValueObject representing a Post author, we can replace and update our Post entity.

At the same time, we should also change the accessor to get the author. Instead of prefixing with “get” we should drop this because we want to use the actual domain language, and we are not really “getting” the author - it is already there. This makes our entity clearer: we have a Post with an Author that has a name and an email.

```

class PostContent
{
    private $content;

    public function __construct($content)
    {
        Assert::that($content, null, 'content')
            ->notEmpty()->minLength(100)->maxLength(65000)
            ;

        $this->content = $content;
    }

    public function __toString()
    {
        return $this->toString();
    }

    public function toString()
    {
        return (string)$this->content;
    }

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

    public function text()
    {
        return strip_tags($this->html());
    }

    public function summary($length = 40)
    {
        return implode(' ', array_slice(explode(' ', $this->text()),
0, $length)) . '...';
    }

    public function equals($test)
    {
        if (__CLASS__ === get_class($test)) {
            return ((string)$test === (string)$this);
        }

        return false;
    }
}

```

## MORE VALUE OBJECTS

- What about other properties?
- Title
  - Group title and slug
  - Protect slug from user
- Content
  - Encapsulate the body
  - Add transforms e.g.:
    - summary
    - text (plain text)

© 2017 David Redfern

### More ValueObjects

We could replace the content with another value object that allows us to add extra capabilities such as generating a summary, obtaining a plain text version, or other information. Care must be taken to ensure that the VO only returns new values and does not modify itself.

We could replace the slug and title with a single VO (PostDetails for example - or another name) or individual VOs for each property. Then again, we could simply not have a “slug” and derive it automatically when the title changes. This may be a better option. A Str helper or Slug creator could be used directly in the VO. This would be a matter of preference and design of your domain: is the slug user editable, or automatically generated? There are pros and cons to both approaches.



# TOWARDS A CLEAN API

---

*Steps to make our Entity even better*



## BETTER DATA METHODS

---

```
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;
    }
}
```

- Drop “get” prefixes
- Use domain language
- Return useful data
- Add state methods
  - isPublished
  - isRecentlyPublished
  - hasComments
  - leaveComment

© 2017 David Redfern

### Naming Things is Hard

As for any other methods, rather than using “getters”, the methods can be renamed to use the actual domain language and ensure that we are returning appropriate objects. Dates should always be DateTimeImmutable objects (for example) and we can add additional methods to prevent domain state from being rediscovered. This should allow any view logic to be vastly simplified\*.

The end goal is to have an expressive entity that encompasses our business logic, where it makes sense. If you find you are adding external helpers or writing many if/else statements to check the domain state, you may want to push that logic down into the entity instead.

\* of course you are not passing pure Domain Objects to your view are you?! You are passing ViewModels that encapsulate exactly what the view needs... if you're not, you should maybe consider it!

## SPECIAL CASE

---

- Internal collections
- Mutable
- Don't expose mutability
- Wrap collections
  - Very important with Doctrine
  - Don't allow changes outside of the entity

```
class Post
{
    public function comments()
    {
        return new ArrayCollection($this->comments->toArray());
    }

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

© 2017 David Redfern

### Dealing With Collections

For collections of child entities, we need to take extra precautions. By default the internal Collection is mutable, but we do not want to allow external processes to change this without running through our state change methods.

Can guard against this by returning new Collections detached from the main internal Collection (for Doctrine could be a PersistentCollection instance). If we are using Doctrine, then we could translate to a non-Doctrine collection or use the basic ArrayCollection. Note that for large collections, you would need to implement the Doctrine Criteria or only partially load the internal collections.

Why guard against Doctrine?

Multiple reasons: certain operations can be executed outside of the UoW flush e.g.: many-to-many clear() will cause a delete outside of a transaction. Ultimately we want to enforce data integrity, if the collection can be freely modified, invalid entities could be associated and by-pass our rules.

We are skipping the issue with having free access to the sub-entities contained in the detached collection. if these allow modification we have to be very careful to capture those changes. In these instances what we could do is simply remove all modifying methods and require the parent entity to make changes - or ensure there is a link back to the parent entity and then link modifications back.

### Doctrine?

Doctrine is an Entity Mapper; each entity has a configuration detailing how the entity is persisted to a data-source. Entities are managed by an Entity Manager that connects to a single resource (database). Unlike ActiveRecord, our entity is largely disconnected from the persistence and our entity properties can be very different (name wise and type) to what is stored in the persistence layer.

Like with any ORM there are always trade-offs. Doctrine has a very high upfront cost due to the configuration and wiring and does require some concessions in the entities to make it easier to work with; however in any large project entity mapping will always prove to be a better system than ActiveRecord implementations.





# BROADCASTING CHANGES

---

*De-coupling processing*

# DOMAIN EVENTS

---

- Reflect important changes
- Propagate those changes
- Separate responsibilities
- Cross process data sharing
  - update search indexes
  - send messages
  - more domain changes
- First step to Event Sourcing

```
class Post
{
    private $events = [];

    public static function create(PostAuthor $author, PostTitle
$title, PostContent $content)
    {
        $entity = new static($author, $title, $content);
        $entity->raise(new PostCreated([
            'author' => $author,
            'title' => $title,
            'summary' => $content->summary(),
            'created_at' => $entity->createdAt(),
        ]));

        return $entity;
    }

    protected function raise(Event $event)
    {
        $this->events[] = $event;
    }

    public function releaseEvents()
    {
        $events = $this->events;
        $this->events = [];

        return $events;
    }
}
```

© 2017 David Redfern

## Domain Events

Domain Events are internal events that are created when the domains state changes. Typically these are created by the Aggregate Root and then propagated through an event dispatcher or message queue (e.g. RabbitMQ). This allows many external processes to respond to changes in the domain, without having to be coupled to the language the event came from, or have in-depth domain knowledge.

AggregateRoot represents a master object that ultimately governs the changes to a set of related sub-entities. In essence it is a transaction boundary that ensures all the entities it is composed of are valid. External objects should only link against this aggregate root and never the internal entities. (see: [https://martinfowler.com/bliki/DDD\\_Aggregate.html](https://martinfowler.com/bliki/DDD_Aggregate.html))

Consider in our Post example, when the Post is published we may want to:

- \* notify subscribers
- \* update search indexes
- \* update RSS feeds etc.

By using domain events we can broadcast that the post was published, with the headline data and allow all those other processes to run without requiring domain knowledge. Those processes could be written in any language and run on any server, allowing us to truly scale out our application.

We can even have domain events that trigger more domain events, cascading domain changes.

Of course: many events can make it harder to debug... everything is trade-offs.

```

class DomainEventPublisher implements EventSubscriber
{
    private $entities;

    // abbreviated code...
    public function __construct()
    {
        $this->entities = new Collection();
    }

    public function getSubscribedEvents()
    {
        return [
            Events::prePersist, Events::preFlush, Events::postFlush
        ];
    }

    public function prePersist(LifecycleEventArgs $event)
    {
        $entity = $event->getEntity();

        if ($entity instanceof RaisesDomainEvents) {
            $this->entities->add($entity);
        }
    }

    public function preFlush(PreFlushEventArgs $event)
    {
        $uow = $event->getEntityManager()->getUnitOfWork();

        foreach ($uow->getIdentityMap() as $class => $entities) {
            foreach ($entities as $entity) {
                $this->entities->add($entity);
            }
        }
    }

    public function postFlush(PostFlushEventArgs $event)
    {
        // dispatch events, add other channels here
        $events->call(function ($event) use ($sem, $sevm) {
            $sevm->dispatchEvent('on' . $event->name(), $event);
        });

        $this->entities->reset();
    }
}

```

# DISPATCHING EVENTS

- Use framework dispatcher
- Can use Doctrine
  - Listen for:
    - prePersist
    - preFlush
    - postFlush
  - Publish events post flush
- Can broadcast via RabbitMQ

© 2017 David Redfern

## Dispatching Events

See: <https://github.com/dave-redfern/better-entities/blob/master/src/Support/Doctrine/Subscribers/DomainEventPublisher.php>

We can use Doctrine to dispatch the Domain Events after the UoW has been flushed. Benefit here is that we have fully persisted entities and can obtain any persistence created identities. Code above is very very simplified, see the Github repo for the full implementation.

Because of the way Doctrine works internally, we need to hook into both prePersist AND preFlush. New entities do not trigger the preFlush event, so have to be handled separately by the prePersist. We collect all the entities that fire domain events (via interface detection) and then wait for the postFlush to be triggered.

First step is to ensure the events are fully ordered in create order: this is why the domain events use the micro-time as float to be able to get micro-second precision. The events are then ordered using a bcmath compare function with 6 digit precision. The events can then be broadcast through Doctrines internal dispatcher or to any other dispatch channel.

If using Doctrines internal dispatcher, then the domain event needs to inherit from Doctrine\Common\EventArgs, however we could inject any event dispatcher or queue system and not use Doctrine at all.

A common usage is to inject a RabbitMQ connection instance and then broadcast the domain event to a fan-out arrangement on a dedicated channel purely for that. Routing keys become important if you opt for this approach, as does persisting the RabbitMQ queues.

```
class Post implements RaisesDomainEventsContract
{
    use RaisesDomainEvents;

    private $id, $author, $title, $content, $createdAt, $updatedAt, $publishedAt, $comments;
    private function __construct(PostAuthor $author, PostTitle $title, PostContent $content)
    {
        $this->author    = $author;
        $this->title     = $title;
        $this->content    = $content;
        $this->createdAt = new DateTimeImmutable();
        $this->updatedAt = new DateTimeImmutable();
        $this->comments  = new ArrayCollection();
    }

    public static function create(PostAuthor $author, PostTitle $title, PostContent $content)
    {
        $entity = new static($author, $title, $content);
        $entity->raise(new Events\PostCreated([
            'author' => $author, 'title' => $title, 'created_at' => $entity->createdAt(),
        ]));
        return $entity;
    }

    public static function createAndPublish(PostAuthor $author, PostTitle $title, PostContent $content)
    {
        $entity = static::create($author, $title, $content);
        $entity->publish();
        return $entity;
    }

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

        $this->raise(new Events\PostPublished([
            'author' => $this->author, 'title' => $this->title, 'published_at' => $this->publishedAt(),
        ]));
    }

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

        $this->raise(new Events\PostRemovedFromPublishedList([
            'author' => $this->author, 'title' => $this->title, 'removed_at' => new DateTimeImmutable(),
        ]));
    }

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

        $this->raise(new Events\PostTitleChanged([
            'author' => $this->author, 'title' => $this->title, 'updated_at' => new DateTimeImmutable(),
        ]));
    }

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

        $this->raise(new Events\PostContentChanged([
            'author' => $this->author, 'title' => $this->title, 'updated_at' => new DateTimeImmutable(),
        ]));
    }

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

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

        $this->raise(new Events\CommentLeftOnPost([
            'title'      => $this->title,
            'commenter'  => $commenter,
            'comment'    => $comment,
            'created_at' => new DateTimeImmutable(),
        ]));
    }
}
```

# CLEAN, RICH, DOMAIN OBJECTS

*(<https://github.com/dave-redfern/better-entities/>) The End Result*

### The End Result

By the end of this exercise we have transformed out entities to use our domain language, enforce specific state changes, allow capturing those changes and broadcasting them while adding a clean API with strict type checking.

Checkout the Github repo to see what can be done. Our post could end up looking like:

<https://github.com/dave-redfern/better-entities/blob/master/src/Entities/Post.php>



## MORE...

---

- Domain Driven Design
  - Eric Evans (blue book)
  - Vernon Vaughn (red book)
- Event Sourcing
  - Greg Young
- Doctrine Project
- Example code available
  - <https://github.com/dave-redfern/better-entities>

© 2017 David Redfern

### Example Code

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