Introduction to Event Sourcing and CQRS

`git.io/vUb0C`

https://github.com/dflydev/es-cqrs-broadway-workshop

Beau Simensen <@beausimensen>
Willem-Jan Zijderveld <@willemjanz>

joind.in/14190

Strategy is hugely important!

But today we'll be looking at tactics...

My Story

Domain-Driven Design and finding "purity"

(I believe the latter has contributed greatly to my occasional grumpiness)

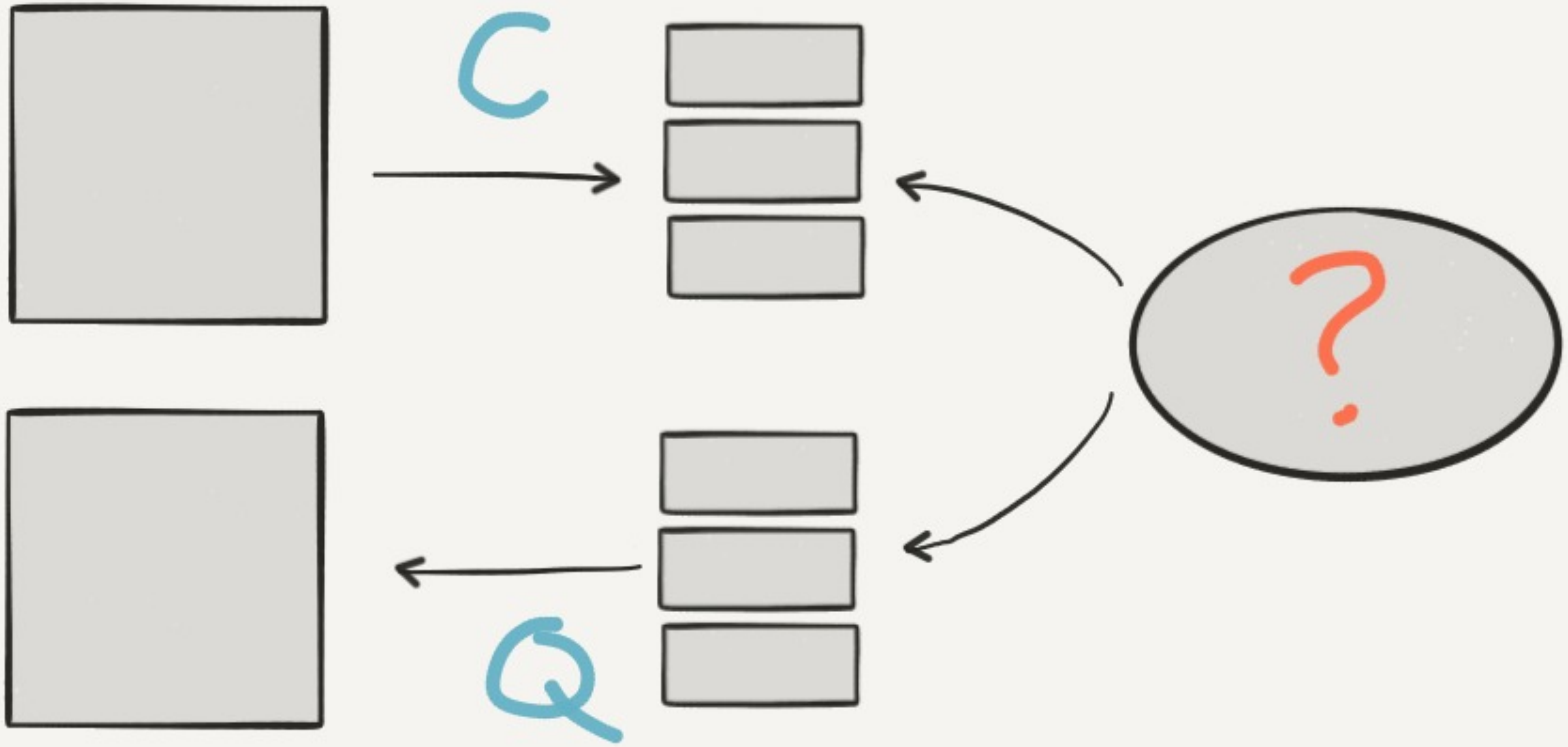
I was stuck in the land of

persisting last known state

Event Sourcing & CQRS

CQRS

Command / Query
Responsibility Segregation


C
Q
?

Our Model

```
class Post
{
    /** @var string */
    private $id;

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

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

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

    /** @var bool[] */
    private $tags = [];

    /** @var string */
    private $status;
}
```

```php
class Post
{
    public function __construct($id) { $this->id = $id; }
    public function getId() { return $this->id; }
    public function getTitle() { return $this->title; }
    public function getContent() { return $this->content; }
    public function getCategory() { return $this->category; }
    public function getTags() {
        return array_keys($this->tags);
    }
}
```

```php
class Post
{
    public function publish($title, $content, $category) {
        $this->title = $title;
        $this->content = $content;
        $this->category = $category;
    }

    public function addTag($tag) {
        $this->tags[$tag] = true;
    }

    public function removeTag($tag) {
        if (isset($this->tags[$tag])) {
            unset($this->tags[$tag]);
        }
    }
}
```

```
interface PostRepository {
    public function find($id);
    public function findAll();
    public function save($post);
}
```

Assumption: This model is "Business Correct"

Back to reality

UI Requirement #1

We MUST be able to see a count of the number of posts with each category.

```
// Raw SQL
SELECT COUNT(*)
  FROM post
 GROUP BY category
```

… or using the (No)SQL-Like language or query builder thingy for your ORM/ODM of choice

UI Requirement #2

We MUST be able to see a count of the number of posts with each tag.

```
// Raw SQL (maybe?)
SELECT COUNT(*)
  FROM post_tags
 WHERE tag = :tag
 GROUP BY tag
```

… since tags is array-ish, this depends quite a bit on the underlying implementation…

```
// Raw SQL (maybe?)
SELECT COUNT(*)
  FROM post_tags
 WHERE tag = :tag
 GROUP BY tag
```

Oh, btw, did you serialize a raw array into your column? You're probably out of luck!

Also… where should this code go?

```php
interface PostRepository
{
    // ...
    public function getNumberOfPostsWithCategory($category);
    public function getNumberOfPostsWithTag($tag);
}
```

Exposing a query builder could turn out to be a lot of work

And would likely leak implementation details

(Think: post_tags)

UI starts to influence the domain model

Optimize for...

Read?

Write?

BOTH!

Introduce

Read Models

with a little help from events

Model -> ??? -> Read Model?

Events describe interesting things that have already happened

Use past tense names

AccountWasCharged, PricingLevelChanged, PilotEjectedFromPlane

What events describe interesting things that have happened to our Post?

```php
class Post
{

    public function publish($title, $content, $category) { /** */ }


    public function addTag($tag) { /** */ }


    public function removeTag($tag) { /** */ }
}
```

```
class Post
{
    // PostWasPublished, PostWasCategorized, PostWasUncategorized
    public function publish($title, $content, $category) { /** */ }


    public function addTag($tag) { /** */ }


    public function removeTag($tag) { /** */ }
}
```

```
class Post
{
    // PostWasPublished, PostWasCategorized, PostWasUncategorized
    public function publish($title, $content, $category) { /** */ }

    // PostWasTagged
    public function addTag($tag) { /** */ }


    public function removeTag($tag) { /** */ }
}
```

```php
class Post
{
    // PostWasPublished, PostWasCategorized, PostWasUncategorized
    public function publish($title, $content, $category) { /** */ }

    // PostWasTagged
    public function addTag($tag) { /** */ }

    // PostWasUntagged
    public function removeTag($tag) { /** */ }
}
```

```
interface RecordsEvents {
    public function getRecordedEvents();
}
```

```
class Post implements RecordsEvents {

    //
    // Could be implemented as a base class / trait
    //

    private $recordedEvents = [];

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

    protected function recordEvent($event) {
        $this->recordedEvents[] = $event;
    }
}
```

```php
class Post {
    public function addTag($tag) {
        if (isset($this->tags[$tag])) {
            return;
        }

        $this->tags[$tag] = true;

        $this->recordEvent(new PostWasTagged(
            $this->id,
            $tag
        ));
    }
}
```

```php
class Post {
    public function removeTag($tag) {
        if (! isset($this->tags[$tag])) {
            return;
        }

        unset($this->tags[$tag]);

        $this->recordEvent(new PostWasUntagged(
            $this->id,
            $tag
        ));
    }
}
```

```
class Post {
    public function publish($title, $content, $category) {
        $this->uncategorizeIfCategoryChanged($category);


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

    private uncategorizeIfCategoryChanged($category) {
        if ($category === $this->category || ! $this->category) { return; }

        $this->recordEvent(new PostWasUncategorized(
            $this->id,
            $this->category
        ));
    }
}
```

```
class Post {
    public function publish($title, $content, $category) {
        $this->uncategorizeIfCategoryChanged($category);
        $this->categorizeIfCatagoryChanged($category);

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

    private categorizeIfCatagoryChanged($category) {
        if ($category === $this->category) { return; }

        $this->recordEvent(new PostWasCategorized(
            $this->id,
            $this->category
        ));
    }
}
```

Model -> Events -> ??? -> Read Model?

The Goal

Every time an object is saved its recorded events are dispatched

Event Bus

(… or event dispatcher or whatever)

Infrastructure Listener

```
use Doctrine\Common\EventSubscriber;

class DoctrinePostSubscriber implements EventSubscriber
{
    private $eventBus;

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

    public function getSubscribedEvents() { return ['postPersist']; }

    public function postPersist(EventArgs $eventArgs) {
        $object = $eventArgs->getObject();
        if ($object instanceof RecordsEvents) {
            $this->eventBus->dispatchAll($object->getRecordedEvents());
        }
    }
}
```

```
Post::saving(function (Post $post) use ($eventBus) {
    $eventBus->dispatchAll($post->getRecordedEvents());
});
```

Repository

```php
class SomePostRepository implements PostRepository {
    private $eventBus;
    public function __construct(/** ... */, $eventBus) {
        // ...
        $this->eventBus = $eventBus;
    }
    public function save($post) {
        // ...
        $this->eventBus->dispatchAll($post->getRecordedEvents());
    }
}
```

```php
class RecordedEventDispatchingPostRepository implements PostRepository {
    private $postRepository;
    private $eventBus;
    public function __construct(PostRepository $postRepository, $eventBus) {
        $this->postRepository = $postRepository;
        $this->eventBus = $eventBus;
    }
    public function find($id) { return $this->postRepository->find($id); }
    public function findAll() { return $this->postRepository->findAll(); }
    public function save($post) {
        $this->postRepository->save($post);
        $this->eventBus->dispatchAll($post->getRecordedEvents());
    }
}
```

Model -> Events -> Event Bus -> ??? -> Read Model?

Read Model

```php
class PostTagCount {
    private $tag;
    private $count;
    public function __construct($tag, $count) {
        $this->tag = $tag;
        $this->count = $count;
    }
    public function getTag() { return $this->tag; }
    public function getCount() { return $this->count; }
}
```

```
interface PostTagCountRepository {
    public function find($tag);
    public function findAll();
    public function increment($tag);
    public function decrement($tag);
}
```

```
class RedisPostTagCountRepository implements PostTagCountRepository {
    const KEY = 'post_tag_count';

    private $redis;

    public function __construct($redis) {
        $this->redis = $redis;
    }
}
```

```php
class RedisPostTagCountRepository implements PostTagCountRepository {
    public function increment($tag) {
        $this->redis->hincrby(static::KEY, $tag, 1);
    }

    public function decrement($tag) {
        $this->redis->hincrby(static::KEY, $tag, -1);
    }
}
```

```php
class RedisPostTagCountRepository implements PostTagCountRepository {
    public function find($tag) {
        $count = $this->redis->hget(static::KEY, $tag);
        if (is_null($count)) { return null; }
        return new PostTagCount($tag, $count);
    }

    public function findAll() {
        $results = [];
        foreach ($this->redis->hgetall(static::KEY) as $tag => $count) {
            $results[] = new PostTagCount($tag, $count);
        }
        return $results;
    }
}
```

```php
class PostCategoryCount {
    private $category;
    private $count;
    public function __construct($category, $count) {
        $this->category = $category;
        $this->count = $count;
    }
    public function getCategory() { return $this->category; }
    public function getCount() { return $this->count; }
}
```

```php
interface PostCategoryCountRepository {
    public function find($category);
    public function findAll();
    public function increment($category);
    public function decrement($category);
}
```

```
class EloquentPostCategoryCountRepository implements PostCategoryCountRepository {
    public function find($category) {
        try {
            $post_category_count = PostCategoryCount::firstOrFail([
                'category' => $category,
            ]);

            return new PostCategoryCount(
                $category,
                $post_category_count->category_count
            );
        } catch (\Exception $e) {
            return null;
        }
    }
}
```

```php
class EloquentPostCategoryCountRepository implements PostCategoryCountRepository {
    public function increment($category) {
        DB::transactional(function () {
            $post_category_count = PostCategoryCount::firstOrNew([
                'category' => $category,
            ]);

            $post_category_count->category_count++;
            $post_category_count->save();
        });
    }
}
```

```
class EloquentPostCategoryCountRepository implements PostCategoryCountRepository {
    public function decrement($category) {
        DB::transactional(function () {
            $post_category_count = PostCategoryCount::firstOrNew([
                'category' => $category,
            ]);

            $post_category_count->category_count--;
            $post_category_count->save();
        });
    }
}
```

Read Model is not bound in any way to Model's persistence layer

(though it could be...)

Read Model can be optimized for speed and specific query requirements

Model -> Events -> Event Bus -> ??? -> Read Model!

Projector

```
interface Projector {
    public function handle($event);
}
```

```php
abstract class ConventionBasedProjector implements Projector {
    public function handle($event) {
        $method = $this->getHandleMethod($event);

        if (! method_exists($this, $method)) { return; }

        $this->$method($event, $event);
    }

    private function getHandleMethod($event) {
        $classParts = explode('\\', get_class($event));

        return 'apply' . end($classParts);
    }
}
```

```php
class PostCategoryCountProjector extends ConventionBasedProjector {
    private $repository;

    public function __construct(PostCategoryCountRepository $repository) {
        $this->repository = $repository;
    }

    public function applyPostWasCategorized(PostWasCategorized $event) {
        $this->repository->increment($event->category);
    }

    public function applyPostWasUncategorized(PostWasUncategorized $event) {
        $this->repository->decrement($event->category);
    }
}
```

Model -> Events -> Event Bus -> Projector -> Read Model!

You *could* stop here…

We've augmented state based Model persistence with event driven Read Models to account for specialized query requirements

But have we achieved

Command / Query Segregation?

Not really…

We've created a Read Model
... but remember these getters?

```php
class Post
{
    public function getId() { return $this->id; }
    public function getTitle() { return $this->title; }
    public function getContent() { return $this->content; }
    public function getCategory() { return $this->category; }
    public function getTags() {
        return array_keys($this->tags);
    }
}
```

```php
class Post
{
    public function getId() { return $this->id; }
    //public function getTitle() { return $this->title; }
    //public function getContent() { return $this->content; }
    //public function getCategory() { return $this->category; }
    //public function getTags() {
    //     return array_keys($this->tags);
    //}
}
```

?!?!?!

Time to introduce
Yet Another Read Model

```php
class PublishedPost {
    public $id;
    public $title;
    public $content;
    public $category;
    public function __construct($id) {
        $this->id = $id;
    }
}
```

```
interface PublishedPostRepository {
    public function find($id);
    public function findAll();
    public function save($publishedPost);
}
```

```php
class Post {
    public function publish($title, $content, $category) {
        $this->uncategorizeIfCategoryChanged($category);
        $this->categorizeIfCatagoryChanged($category);

        $this->title = $title;
        $this->content = $content;
        $this->category = $category;

        $this->recordEvent(new PostWasPublished(
           $this->id,
           $title,
           $content,
           $category
        ));
    }
}
```

```php
class PublishedPostProjector extends ConventionBasedProjector {
    private $repository;

    public function __construct(PublishedPostRepository $repository) {
        $this->repository = $repository;
    }

    public function applyPostWasPublished(PostWasPublished $event) {
        $publishedPost = $this->repository->find($event->id);
        $publishedPost->title = $event->title;
        $publishedPost->content = $event->content;
        $publishedPost->category = $event->category;
        $this->repository->save($publishedPost);
    }
}
```

Why would this be problematic?

```
class Post {
    public function __construct($id) {
        $this->id = $id;
        $this->recordEvent(new PostWasCreated($ie));
    }
}
```

Every time a new Post is instantiated it would result in recording a new PostWasCreated event

(not really what we are going for here…)

```php
class Post {
    public static function create($id) {
        $instance = new static($id);
        $instance->recordEvent(new PostWasCreated($id));

        return $instance;
    }
}
```

```php
class PublishedPostProjector extends ConventionBasedProjector {
    public function applyPostWasCreated(PostWasCreated $event) {
        $publishedPost = new PublishedPost($event->id);
        $this->repository->save($publishedPost);
    }

    public function applyPostWasPublished(PostWasPublished $event) {
        $publishedPost = $this->repository->find($event->id);
        $publishedPost->title = $event->title;
        $publishedPost->content = $event->content;
        $publishedPost->category = $event->category;
        $this->repository->save($publishedPost);
    }
}
```

Impact on a controller?

```
// before...
Route::get('/post/{id}', function ($id) {
    return view('post')->withPost(
        $postRepository->find($id)
    );
});

// after...
Route::get('/post/{id}', function ($id) {
    return view('post')->withPost(
        $publishedPostRepository->find($id);
    );
});
```

So have we now achieved

Command / Query Segregation?

Yes!

But there is another thing we can do…

Let's make Commands

EXPLICIT

in our domain

Events represent activities that *happened in the past*

Commands represent things that *should happen in the future*

Use imperative names

ChargeAccount, ChangePricingLevel, EjectPilotFromPlane

What do we need to know in order to be able to publish a Post?

```
Route::put('/post/{id}', function ($request, $postRepository, $id) {
    $post = $postRepository->find($id);
    $post->publish($request->title, $request->content, $request->category);
    $postRepository->save($post);

    return view('post.published');
});
```

What name should we use for the Command to publish a Post?

```
Route::put('/post/{id}', function ($request, $postRepository, $id) {
    $post = $postRepository->find($id);
    $post->publish($request->title, $request->content, $request->category);
    $postRepository->save($post);

    return view('post.published');
});
```

```
class PublishPost {
    public $id;
    public $title;
    public $content;
    public $category;
    public function __construct($id, $title, $content, $category) {
        $this->id = $id;
        $this->title = $title;
        $this->content = $content;
        $this->category = $category;
    }
}
```

Command -> ??? -> Model -> Events -> Event Bus -> Projector -> Read Model!

Command Bus

Similar to Event Bus

But used for Commands :)

```
Route::put('/post/{id}', function ($request, $commandBus, $id) {
    // $post = $postRepository->find($id);
    // $post->publish($request->title, $request->content, $request->category);
    // $postRepository->save($post);
    $commandBus->dispatch(new PublishPost(
        $id,
        $request->title,
        $request->content,
        $request->category
    ));

    return view('post.published');
});
```

Command -> Command Bus -> ??? -> Model -> Events -> Event Bus -> Projector -> Read Model!

Command Handler

Responsible for running the Command on the model

Only one Command Handler for each Command

```php
interface CommandHandler {
    public function handle($command);
}
```

```
abstract class ConventionBasedCommandHandler implements CommandHandler
{
    public function handle($command)
    {
        $method = $this->getHandleMethod($command);

        if (! method_exists($this, $method)) { return; }

        $this->$method($command);
    }

    private function getHandleMethod($command)
    {
        if (! is_object($command)) {
            throw new CommandNotAnObjectException();
        }

        $classParts = explode('\\', get_class($command));

        return 'handle' . end($classParts);
    }
}
```

```php
class PublishPostHandler extends ConventionBasedCommandHandler {
    private $postRepository;
    public function __construct($postRepository) {
        $this->postRepository = $postRepository;
    }
    public function handlePublishPost(PublishPost $command) {
        $post = $this->postRepository->find($command->id);
        $post->publish(
            $command->title,
            $command->content,
            $command->category
        );
        $this->postRepository->save($post);
    }
}
```

```
Route::put('/post/{id}', function ($request, $commandBus, $id) {
    $commandBus->dispatch(new PublishPost(
        $id,
        $request->title,
        $request->content,
        $request->category
    ));

    return view('post.published');
});

class PublishPostHandler {
    public function handlePublishPost(PublishPost $command) {
        $post = $this->postRepository->find($command->id);
        $post->publish(
            $command->title,
            $command->content,
            $command->category
        );
        $this->postRepository->save($post);
    }
}
```

Command -> Command Bus -> Command Handler -> Model -> Events -> Event Bus -> Projector -> Read Model!

What is our stateful model doing for us?

Keep in mind that Post no longer has getters!

Is state important here?

```
class Post {
    public function addTag($tag) {
        if (isset($this->tags[$tag])) {
            return;
        }

        $this->tags[$tag] = true;

        $this->recordEvent(new PostWasTagged(
            $this->id,
            $tag
        ));
    }
}
```

Is state important here?

```
class Post {
    public function removeTag($tag) {
        if (! isset($this->tags[$tag])) {
            return;
        }

        unset($this->tags[$tag]);

        $this->recordEvent(new PostWasUntagged(
            $this->id,
            $tag
        ));
    }
}
```

Is state important here?

```php
class Post {
    public function __construct($id) {
        $this->id = $id;
    }
}
```

Is state important here?

```
class Post {
    public function publish($title, $content, $category) {
        $this->uncategorizeIfCategoryChanged($category);
        $this->categorizeIfCatagoryChanged($category);

        $this->title = $title;
        $this->content = $content;
        $this->category = $category;

        $event = new PostWasPublished($this->id, $title, $content, $category);

        $this->recordEvent($event);
    }
}
```

Is state important here?

```
class Post {
    public function publish($title, $content, $category) {
        $this->uncategorizeIfCategoryChanged($category);
        $this->categorizeIfCatagoryChanged($category);

        //$this->title = $title;
        //$this->content = $content;
        $this->category = $category;

        $event = new PostWasPublished($this->id, $title, $content, $category);

        $this->recordEvent($event);
    }
}
```

We have the data in the read model...

But the read model data should be considered volatile

What if we find a bug in the projections?
Our source of truth would be tainted.

What if Redis crashes?
We lose the data altogether.

Potential solution?

What if we store all of the published events...

Potential solution?

… so we could replay them through the projectors if needed?

Potential solution?

… which would mean we could replay them through NEW projectors?

Potential solution?

... wouldn't that mean we should be able to rebuild the *model itself* from past events?

Command -> Command Bus -> Command Handler -> Model -> Events -> ??? -> Event Store -> Event Bus -> Projector -> Read Model!

Step 1

Make Post capable of handling events

```php
class Post {
    protected function handle($event) {
        $method = $this->getHandleMethod($event);

        if (! method_exists($this, $method)) { return; }

        $this->$method($event);
    }

    private function getHandleMethod($event) {
        $classParts = explode('\\', get_class($event));

        return 'apply' . end($classParts);
    }
}
```

Step 2

Make recordEvent() handle events

(ideally we'd rename this method)

```php
class Post {
    protected function recordEvent($event) {
        $this->handle($event);
        $this->recordedEvents[] = $event;
    }
}
```

Step 3

Move state changes into event handler methods

```php
class Post {
    public function __construct($id) {
        $this->id = $id;
    }




}
```

```php
class Post {
    private function __construct() {
        // $this->id = $id;
    }

    private function applyPostWasCreated(PostWasCreated $event) {
        $this->id = $event->id;
    }
}
```

```php
class Post {
    public function addTag($tag) {
        if (isset($this->tags[$tag])) {
            return;
        }

        $this->tags[$tag] = true;

        $this->recordEvent(new PostWasTagged(
            $this->id,
            $tag
        ));
    }




}
```

```php
class Post {
    public function addTag($tag) {
        if (isset($this->tags[$tag])) {
            return;
        }

        // $this->tags[$tag] = true;

        $this->recordEvent(new PostWasTagged(
            $this->id,
            $tag
        ));
    }

    private function applyPostWasTagged(PostWasTagged $event) {
        $this->tags[$event->tag] = true;
    }
}
```

```
class Post {
    public function removeTag($tag) {
        if (! isset($this->tags[$tag])) {
            return;
        }

        unset($this->tags[$tag]);

        $this->recordEvent(new PostWasUntagged(
            $this->id,
            $tag
        ));
    }




}
```

```php
class Post {
    public function removeTag($tag) {
        if (! isset($this->tags[$tag])) {
            return;
        }

        // unset($this->tags[$tag]);

        $this->recordEvent(new PostWasUntagged(
            $this->id,
            $tag
        ));
    }

    private function applyPostWasUntagged(PostWasUntagged $event) {
        unset($this->tags[$event->tag]);
    }
}
```

```php
class Post {
    public function publish($title, $content, $category) {
        $this->uncategorizeIfCategoryChanged($category);
        $this->categorizeIfCatagoryChanged($category);

        $this->title = $title;
        $this->content = $content;
        $this->category = $category;

        $this->recordEvent(new PostWasPublished(
            $this->id,
            $title,
            $content,
            $category
        ));
    }




}
```

```php
class Post {
    public function publish($title, $content, $category) {
        $this->uncategorizeIfCategoryChanged($category);
        $this->categorizeIfCatagoryChanged($category);

        //$this->title = $title;
        //$this->content = $content;
        //$this->category = $category;

        $this->recordEvent(new PostWasPublished(
            $this->id,
            $title,
            $content,
            $category
        ));
    }

    private function applyPostWasUntagged(PostWasUntagged $event) {
        $this->category = $event->category;
    }
}
```

Step 4

Initializing State from previously recorded events

```php
interface AppliesRecordedEvents {
    public function applyRecordedEvents(array $events);
}
```

```php
class Post implements AppliesRecordedEvents {
    public function applyRecordedEvents(array $events) {
        foreach ($events as $event) {
            $this->handle($event);
        }
    }
}
```

Oh noes!

How do we instantiate a new instance without specifying an ID?

```php
class Post {
    private function __construct() { }




}
```

```php
class Post {
    private function __construct() { }

    public static function instantiateForReconstitution() {
        return new static();
    }
}
```

```php
$post1 = Post::create(1);
$post1->publish('hello', 'world', 'draft');
$post1->addTag('es');
$post1->addTag('cqrs');
$post1->removeTag('es');

$recordedEvents = $post1->getRecordedEvents();

// $recordedEvents = [
//     new PostWasCreated(1),
//     new PostWasPublished(1, 'hello', 'world', 'draft'),
//     new PostWasTagged(1, 'es',
//     new PostWasTagged(1, 'cqrs'),
//     new PostWasUntagged(1, 'es'),
// ];

$post2 = Post::instantiateForReconstitution();
$post2->applyRecordedEvents($recordedEvents);
```

Event Store

Load existing events and append new events

Identity

An event stream should exist for each object

```php
interface EventStore {
    /**
     * @param  string $identity
     * @return array  Previously recorded events
     */
    public function load($identity);

    /**
     * @param  string $identity
     * @param  array  Newly recorded events
     * @return void
     */
    public function append($identity, array $events);
}
```

WARNING

This is an extremely over simplified event store interfaces

```php
$post = Post::create('some-identity');
$post->publish('hello', 'world', 'draft');
$post->addTag('es');
$post->addTag('cqrs');
$post->removeTag('es');

$recordedEvents = $post->getRecordedEvents();

$eventStore->append(
    'some-identity',
    $recordedEvents
);
```

```
$recordedEvents = $eventStore->load('some-identity');
$post = Post::instantiateForReconstitution();
$post->applyRecordedEvents($recordedEvents);
```

```
EventStoreAndDispatchingPostRepository implements PostRepository
{
    public function __construct($eventStore, $eventBus) {
        $this->eventStore = $eventStore;
        $this->eventBus = $eventBus;
    }
}
```

```
EventStoreAndDispatchingPostRepository implements PostRepository
{
    public function save(Post $post) {
        $recordedEvents = $post->getRecordedEvents();

        $this->eventStore->append(
            $post->getId(),
            $recordedEvents
        );

        $this->eventBus->dispatchAll($recordedEvents);
    }
}
```

```php
EventStoreAndDispatchingPostRepository implements PostRepository
{
    public function find($id) {
        $recordedEvents = $this->eventStore->load($id);
        $post = Post::instantiateForReconstitution();
        $post->applyRecordedEvents($recordedEvents);

        return $post;
    }

    public function findAll() {
        // ???
    }
}
```

Queries will likely be slow

So operations like `findAll()` may be problematic depending on Event Store implementation

Take advantage of CQRS

Queries like "find all" should be coming from your read model anyway!

```
interface PostRepository {
    public function find($id);
    public function findAll();
    public function save($post);
}
```

```php
interface PostRepository {
    public function find($id);
    //public function findAll();
    public function save($post);
}
```

Command -> Command Bus -> Command Handler -> Model -> Events -> Event Store -> Event Bus -> Projector -> Read Model!

So where are we *really*?

Basic framework for
Event Sourcing & CQRS

We have great building blocks!

But we are still missing a few *critical* pieces…

CQRS

We have no Command Bus or Event Bus

Event Sourcing

We have no Event Bus or Event Store

"I'm out."

— People when they realize how much work it takes to build a proper Event Store.

But we can still TEST it!

Given, When, Then.

Testing commands and event streams

Model Scenarios

Testing model event streams

```php
$this->scenario

    ->given([
        new PostWasCreated($id),
        new PostWasCategorized($id, 'news'),
        new PostWasPublished($id, 'title', 'content', 'news'),
        new PostWasTagged($id, 'event-sourcing'),
        new PostWasTagged($id, 'broadway'),
    ])

    ->when(function (Post $post) {
        $post->addTag('cqrs');
    })

    ->then([
        new PostWasTagged($id, 'cqrs'),
    ])

;
```

```php
class PostScenario {
    public function __construct(TestCase $testCase) {
        $this->testCase = $testCase;
    }
}
```

```php
class PostScenario {
    public function given(array $givens = []) {
        if (! $givens) {
            $this->post = null;

            return $this;
        }

        $post = Post::instantiateForReconstitution();
        $post->applyRecordedEvents($givens);

        $this->post = $post;

        return $this;
    }
}
```

```
class PostScenario {
    public function when($when) {
        if (! is_callable($when)) {
            return $this;
        }

        if ($this->post) {
            $when($this->post);
        } else {
            $this->post = $when(null);
        }

        return $this;
    }
}
```

```php
class PostScenario {
    public function then(array $thens) {
        $this->testCase->assertEquals(
            $thens,
            $this->post->getRecordedEvents()
        );

        $this->post->clearRecordedEvents();

        return $this;
    }
}
```

Command Handler Scenarios

Testing command handlers and event streams

```php
$this->scenario

    ->given([
        new PostWasCreated($id),
        new PostWasCategorized($id, 'news'),
        new PostWasPublished($id, 'title', 'content', 'news'),
        new PostWasTagged($id, 'event-sourcing'),
        new PostWasTagged($id, 'broadway'),
    ])

    ->when(new TagPost($id, 'cqrs'))

    ->then([
        new PostWasTagged($id, 'cqrs'),
    ])

;
```

```php
class PostHandlerScenario {
    public function __construct(
        TestCase $testCase,
        SpyingEventStore $eventStore,
        $commandHandler
    ) {
        $this->testCase = $testCase;
        $this->eventStore = $eventStore;
        $this->commandHandler = $commandHandler;
    }
}
```

```php
class PostHandlerScenario {
    public function withId($id) {
        $this->id = $id;

        return $this;
    }
}
```

```php
class PostHandlerScenario {
    public function given(array $events = []) {
        if (! $events) {
            return $this;
        }

        foreach ($events as $event) {
            $this->eventStore->appendEvents($this->id, [$event]);
        }

        $this->eventStore->clearRecordedEvents();

        return $this;
    }
}
```

```php
class PostHandlerScenario {
    public function when($command) {
        $this->commandHandler->handle($command);

        return $this;
    }
}
```

```php
class PostHandlerScenario {
    public function then(array $events = []) {
        $this->testCase->assertEquals(
            $events,
            $this->eventStore->getRecordedEvents()
        );

        $this->eventStore->clearRecordedEvents();

        return $this;
    }
}
```

```
abstract class AbstractPostHandlerTest extends \PHPUnit_Framework_TestCase
{
    protected $scenario;

    public function setUp() {
        $eventStore = new SpyingEventStore(new InMemoryEventStore());
        $eventBus = new SimpleEventBus();
        $postRepository = new SuperAwesomePostRepository($eventStore, $eventBus);

        $this->scenario = new PostHandlerScenario(
            $this,
            $eventStore,
            $this->createCommandHandler($postRepository)
        );
    }

    abstract protected function createCommandHandler(PostRepository $postRepository);
}
```

```php
class CreatePostHandlerTest extends AbstractPostHandlerTest
{
    /** @test */
    public function it_can_create() {
        $id = 'my-id';

        $this->scenario
            ->when(new CreatePost($id))
            ->then([
                new PostWasCreated($id),
            ])
        ;
    }

    protected function createCommandHandler(PostRepository $postRepository) {
        return new CreatePostHandler($postRepository);
    }
}
```

About those missing pieces…

Be Practical

Do you need to implement this yourself?

Broadway

labs.qandidate.com

Command from CQRS

Command Handling and Testing

Query from CQRS

Event Handling, Read Model and Testing

Event Sourcing

Event Handling, Event Store and Testing

Domain-Driven Design Friendly

Repositories, Aggregate Roots, Child Entities, and Aggregate Root Testing

Learn more about Broadway in detail this afternoon!

<live coding>

Thanks!

git.io/vUb0C

@thatpodcast

Beau Simensen <@beausimensen>
Willem-Jan Zijderveld <@willemjanz>

joind.in/14190