Introduction to Event Sourcing and CQRS With Broadway

git.io/vUb0C

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

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

joind.in/14200

Event Sourcing & CQRS

Involve a lot of moving pieces

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

No production ES/CQRS solution
(at least when I started looking for one…)

Roll my own?

I tried.

The sample project from Implementing Domain-Driven Design

https://github.com/VaughnVernon/IDDD_Samples

Buttercup.Protects

https://github.com/buttercup-php/protects

Event Centric

https://github.com/event-centric

:(

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

Components

Domain Component

Domain Message
(an envelope for an event)

Aggregate Root

(where we get the concept of identity & uncommitted events)

```
class Post implements AggregateRoot {
    /**
     * @return DomainEventStream
     */
    public function getUncommittedEvents() {
        /** magic! */
    }

    /**
     * @return string
     */
    public function getAggregateRootId() {
        /** we'll implement this. */
    }
}
```

Event Handling Component

Event Bus

```php
interface EventBusInterface
{
    /**
     * Subscribes the event listener to the event bus.
     *
     * @param EventListenerInterface $eventListener
     */
    public function subscribe(EventListenerInterface $eventListener);

    /**
     * Publishes the events from the domain event stream to the listeners.
     *
     * @param DomainEventStreamInterface $domainMessages
     */
    public function publish(DomainEventStreamInterface $domainMessages);
}
```

Event Listener

(to handle a specific `DomainMessage`)

```php
interface EventListenerInterface
{
    /**
     * @param DomainMessage $domainMessage
     */
    public function handle(DomainMessage $domainMessage);
}
```

Command Handling Component

Command Bus

```php
interface CommandBusInterface
{
    /**
     * Dispatches the command $command to the proper CommandHandler
     *
     * @param mixed $command
     */
    public function dispatch($command);

    /**
     * Subscribes the command handler to this CommandBus
     */
    public function subscribe(CommandHandlerInterface $handler);
}
```

Command Handler

Implement the interface directly

```
class CreatePostHandler implements CommandHandlerInterface {
    public function handle($command)
    {
        if (! $command instanceof CreatePost) {
            return;
        }

        $post = Post::create($command->id);

        $this->getPostRepository()->save($post);
    }
}
```

Rely on Broadway's conventions

```php
class PostHandler extends CommandHandler {


    // ... other Post-related command handlers...

    public function handleCreatePost(CreatePost $command)
    {
        $post = Post::create($command->id);

        $this->getPostRepository()->save($post);
    }
}
```

Adapt

```
class CreatePostHandler {
    public function handle(CreatePost $command)
    {
        $post = Post::create($command->id);

        $this->getPostRepository()->save($post);
    }
}
```

Adapt

```
class PostHandler extends CommandHandler {
    public function __construct(
        CreatePostHandler $createPostHandler,
        /** ... */
    ) {
        $this->createPostHandler = $createPostHandler;
    }

    public function handleCreatePost(CreatePost $command) {
        $this->createPostHandler->handle($command);
    }
}
```

Your own conventions

```
class CommandHandler implements CommandHandlerInterface {
    public function handle($command) {
        $class = get_class($command);
        if (! isset($this->mapping[$class])) {
            return;
        }

        $this->mapping[$class]->handle($command);
    }

    public function register(YourHandlerInterface $handler) {
        $this->mappings[$handler->handles()] = $handler;
    }
}
```

Command Scenario
(for testing commands)

Given, When, Then.

– Broadway scenarios

```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
abstract class PostHandlerTest extends CommandHandlerScenarioTestCase
{
    protected function createCommandHandler(
        EventStoreInterface $eventStore,
        EventBusInterface $eventBus
    ) {
        $postRepository = BroadwayPostRepository::create(
            $eventStore,
            $eventBus
        );

        return new BroadwayPostCommandHandler(
            new CreatePostHandler($postRepository),
            new PublishPostHandler($postRepository),
            new TagPostHandler($postRepository),
            new UntagPostHandler($postRepository)
        );
    }
}
```

```php
class TagPostHandlerTest extends PostHandlerTest
{
    public function testPostTag() {
        $id = 'my-id';

        $this->scenario
            ->withAggregateId($id)
            ->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'),
            ])
        ;
    }
}
```

Read Model Component

Read Model

```php
interface ReadModelInterface {
    /**
     * @return string
     */
    public function getId();
}

interface SerializableInterface {
    /**
     * @return mixed The object instance
     */
    public static function deserialize(array $data);

    /**
     * @return array
     */
    public function serialize();
}
```

Repository

```php
interface RepositoryInterface
{
    public function save(ReadModelInterface $data);
    public function find($id);
    public function findBy(array $fields);
    public function findAll();
    public function remove($id);
}
```

ElasticSearch Repository

Projector

Projectors are just event listeners

```
interface ProjectorInterface extends EventListenerInterface {
}
```

Manage Read Models

```
class BroadwayPostCategoryCountProjector extends Projector
{
    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);
    }
}
```

Read Model Scenario
(for testing read models)

Read Model tests

```php
class PostCategoryCountTest extends ReadModelTestCase {
    protected function createReadModel() {
        return new PostCategoryCount('drafts', 15);
    }
}
```

Projector tests

```
class PostCategoryCountProjectorTest extends ProjectorScenarioTestCase {
    protected function createProjector(InMemoryRepository $repository)
    {
        $postRepository = new BroadwayPostCategoryCountRepository($repository);
        $postCategoryCountProjector = new PostCategoryCountProjector($postRepository);

        return new BroadwayPostCategoryCountProjector($postCategoryCountProjector);
    }
}
```

Projector tests

```
class PostCategoryCountProjectorTest extends ProjectorScenarioTestCase {
    public function it_returns_to_zero()
    {
        $this->scenario
            ->given([
                new PostWasCategorized('my-id', 'drafts'),
            ])
            ->when(new PostWasUncategorized('my-id', 'drafts'))
            ->then([
                new PostCategoryCount('drafts', 0),
            ])
        ;
    }
}
```

Projector tests

```
class PostCategoryCountProjectorTest extends ProjectorScenarioTestCase {
    public function it_returns_to_zero()
    {
        $this->scenario
            ->given([
                new PostWasCategorized('my-id', 'drafts'),
                new PostWasUncategorized('my-id', 'drafts'),
            ])
            ->then([
                new PostCategoryCount('drafts', 0),
            ])
        ;
    }
}
```

For projector tests there is no difference between

given & when

Only use Broadway Read Models when it

makes sense

"Don't use it for anything but basic read/writes"

– Willem-Jan on Broadway Read Models

Event Store Component

Event Store

```php
interface EventStoreInterface
{
    /**
     * @param mixed $id
     *
     * @return DomainEventStreamInterface
     */
    public function load($id);

    /**
     * @param mixed                      $id
     * @param DomainEventStreamInterface $eventStream
     */
    public function append($id, DomainEventStreamInterface $eventStream);
}
```

DBAL Event Store

Mongo Event Store (WIP)

https://github.com/qandidate-labs/broadway/pull/151

(Get) Event Store

https://github.com/dbellettini/broadway-eventstore

Event Sourcing Component

Event Sourced Aggregate Root

```php
class Post extends EventSourcedAggregateRoot {
    public function getAggregateRootId() {
        return (string) $this->id;
    }

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

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

    public function applyPostWasCategorized(PostWasCategorized $event) {
        $this->category = $event->category;
    }
}
```

Event Sourced Entity

```
class Job extends EventSourcedEntity {
    private $jobSeekerId;
    private $jobId;
    private $title;
    private $description;
    public function __construct($jobSeekerId, $jobId, $title, $description)
    {
        $this->jobSeekerId = $jobSeekerId;
        $this->jobId       = $jobId;
        $this->title       = $title;
        $this->description = $description;
    }
}
```

```php
class Job extends EventSourcedEntity {
    public function describe($title, $description)
    {
        $this->apply(new JobWasDescribed(
            $this->jobSeekerId,
            $this->jobId,
            $title,
            $description
        ));
    }

    public function applyJobWasDescribed(JobWasDescribed $event)
    {
        if ($event->jobId !== $this->jobId) {
            return;
        }

        $this->title = $event->title;
        $this->description = $event->description;
    }
}
```

Event Sourcing Repository

```php
class EventSourcingRepository implements RepositoryInterface {
    /**
     * @param EventStoreInterface              $eventStore
     * @param EventBusInterface                $eventBus
     * @param string                           $aggregateClass
     * @param AggregateFactoryInterface        $aggregateFactory
     * @param EventStreamDecoratorInterface[] $eventStreamDecorators
     */
    public function __construct(
        EventStoreInterface $eventStore,
        EventBusInterface $eventBus,
        $aggregateClass,
        AggregateFactoryInterface $aggregateFactory,
        array $eventStreamDecorators = array()
    ) {
        // ...
    }
}
```

```
class EventSourcingRepository implements RepositoryInterface {
    public function save(AggregateRoot $aggregate)
    {
        // maybe we can get generics one day.... ;)
        Assert::isInstanceOf($aggregate, $this->aggregateClass);
        $domainEventStream = $aggregate->getUncommittedEvents();
        $eventStream       = $this->decorateForWrite($aggregate, $domainEventStream);
        $this->eventStore->append($aggregate->getAggregateRootId(), $eventStream);
        $this->eventBus->publish($eventStream);
    }
}
```

Event Stream Decorator

```php
interface EventStreamDecoratorInterface
{
    /**
     * @param string                       $aggregateType
     * @param string                       $aggregateIdentifier
     * @param DomainEventStreamInterface $eventStream
     *
     * @return DomainEventStreamInterface
     */
    public function decorateForWrite(
        $aggregateType,
        $aggregateIdentifier,
        DomainEventStreamInterface $eventStream
    );
}
```

Metadata Enricher

```php
/**
 * Adds extra metadata to already existing metadata.
 */
interface MetadataEnricherInterface
{
    /**
     * @return Metadata
     */
    public function enrich(Metadata $metadata);
}
```

Aggregate Factories

```
class EventSourcingRepository implements RepositoryInterface {
    public function load($id)
    {
        try {
            $domainEventStream = $this->eventStore->load($id);
            return $this->aggregateFactory->create(
                $this->aggregateClass,
                $domainEventStream
            );
        } catch (EventStreamNotFoundException $e) {
            throw AggregateNotFoundException::create($id, $e);
        }
    }
}
```

```php
class PublicConstructorAggregateFactory implements AggregateFactoryInterface
{
    public function create($aggregateClass, DomainEventStreamInterface $domainEventStream)
    {
        $aggregate = new $aggregateClass();
        $aggregate->initializeState($domainEventStream);
        return $aggregate;
    }
}
```

```php
class NamedConstructorAggregateFactory implements AggregateFactoryInterface
{
    public function __construct(
        $staticConstructorMethod = 'instantiateForReconstitution'
    ) {
        $this->staticConstructorMethod = $staticConstructorMethod;
    }

    public function create(
        $aggregateClass,
        DomainEventStreamInterface $domainEventStream
    ) {
        Assert::true(method_exists($aggregateClass, $this->staticConstructorMethod));

        $methodCall = sprintf('%s::%s', $aggregateClass, $this->staticConstructorMethod);
        $aggregate  = call_user_func($methodCall);

        Assert::isInstanceOf($aggregate, $aggregateClass);

        $aggregate->initializeState($domainEventStream);

        return $aggregate;
    }
}
```

Processor Component

Processor

Processors are just event listeners

```
abstract class Processor implements EventListenerInterface
{
    public function handle(DomainMessage $domainMessage)
    {
        $event  = $domainMessage->getPayload();
        $method = $this->getHandleMethod($event);
        if (! method_exists($this, $method)) {
            return;
        }
        $this->$method($event, $domainMessage);
    }

    private function getHandleMethod($event)
    {
        $classParts = explode('\\', get_class($event));
        return 'handle' . end($classParts);
    }
}
```

Our Model

```php
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);
    }
}
```

```
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"

UI Requirement #1

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

UI Requirement #2

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

<live coding>

Thanks!

`git.io/vUb0C`

@thatpodcast

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

joind.in/14200