.com software Follow
Oct 9, 2022 · 4 min read · ✦ · Listen

Save

Event Sourcing in PHP for Beginners

Easy tutorial for beginner PHP developers. Do you want to start with Event Sourcing in PHP? Let me show you.

Image by Cottonbro

> *WARNING about the publication: it is a **Work-In-Progress**. It is going to be updated periodically with major changes.*

Have you always wanted to start **Event Sourcing** but did not know how to get started? This publication is *just for you*.

We will go through a short description of what this technique is about; where and when to use it; we will finish with an example application in Symfony! Are you interested? Then let's start the journey!

CRUD and Event Sourcing differences

As a rule, we are used to the traditional model of storing the state of objects in database tables. Each row is mapped to an object using an Object-Relational-Mapper (ORM) framework (think Eloquent, Doctrine, Propel).

In the traditional model, the database table always holds *the last state* of the object. In the case of Event Sourcing, we store events that occurred upon that object instead.

Bank account example

The simplest example is handling a bank account. Bob deposited $100 one day, then spent $70 on groceries, then $10 more on newspapers. In the traditional model, the account's database row holds a value of $20. In Event Sourcing, we keep three (meaningful 😍) events to recreate the state of the account:

- Account topped up with $100,
- Transaction for $70,
- Transaction for $10.

Pros of using Event Sourcing

- Full audit log,
- Recreate the state of the object's state at any point in time,

- Undo any number of operations,
- Create an optimized view model of the object's state,
- Event-sourced entities are effortless to unit test and debug.

Cons of using Event Sourcing

- Much harder to implement,
- State adjustments are only possible through events,
- Requires a reliable database to hold events,
- Increased storage volume,
- Time and sequence sensitive.

Applying Event Sourcing

Event Sourcing is a superb choice for any web application that requires an audit trail or the ability to split the write model from the read model (CQRS!)

Don't apply ES if your application is mostly CRUD. The complexity is not worth the effort in such a case.

Let's design **the contract** of the functionality.

An `EventSourceEntityInterface` domain model will record `EventInterface` events to an `EventCollectionInterface` collection.

During model persist, the `EventSourceManagerInterface` service will collect recorded `EventInterface` events from the `EventCollectionInterface` collection and persist them in a storage using the `EventSourceStoreInterface`.

During model retrieve, the `EventSourceManagerInterface` service will load related `EventInterface` events from the `EventSourceStoreInterface` and apply them to the `EventSourceEntityInterface` to reconstitute the model.

The `EventInterface` is self-contained. It is able to normalize its value to an array and reconstitute itself from one.

I have created two additional interfaces to reference a class name of an event source entity and its identifier.

The contract of an event:

```php
<?php

declare(strict_types=1);

namespace App\Es\Contract;

/**
 * Recorded ES event.
 *
 * @psalm-immutable
 */
interface EventInterface
{
    /**
     * Recreate this object from an array.
     */
    public static function fromArray(array $eventData): EventInterface;

    /**
     * Convert stored object to an array.
     */
    public function toArray(): array;
}
```

es-event-interface.php hosted with ❤ by **GitHub** **view raw**

The event is going to be recorded inside an iterable collection:

```php
<?php

declare(strict_types=1);

namespace App\Es\Contract;

use IteratorAggregate;

/**
 * Collection to hold recorded ES events.
 *
 * @template-extends IteratorAggregate<array-key, EventInterface>
 */
interface EventCollectionInterface extends IteratorAggregate
{
    /**
     * Append an event to this collection.
     */
    public function record(EventInterface $event): void;

    /**
     * Clear this collection and return stored events.
     */
    public function popEvents(): self;
}
```

es-collection-interface.php hosted with ❤ by GitHub view raw

By an event source entity:

```php
<?php

declare(strict_types=1);

namespace App\Es\Contract;

/**
 * An event source domain entity.
 */
interface EventSourceEntityInterface
{
    /**
     * Entity factory.
     *
     * @param iterable<EventInterface> $pastEvents
     */
    public static function create(
        string $identifier,
        EventCollectionInterface $collection,
        iterable $pastEvents
    ): self;

    /**
     * Return the ID of this entity.
     */
    public function getEventSourceIdentifier(): string;

    /**
     * Return the ES event collection and clear the collection.
     */
    public function popEventSourceEvents(): EventCollectionInterface;
}
```

es-entity-interface.php hosted with ❤ by **GitHub** **view raw**

This entity's events are going to be persisted using a persistence manager:

```php
<?php

declare(strict_types=1);

namespace App\Es\Contract;

/**
 * Persistence manager for ES entities.
 */
interface EventSourceManagerInterface
{
    /**
     * Persist an ES entity inside a storage.
     */
    public function persist(
        EventSourceEntityInterface $entity
    ): void;

    /**
     * Reconstitute an ES entity from a storage.
     *
     * @template TObject of EventSourceEntityInterface
     *
     * @param class-string<TObject> $entityClass
     */
    public function reconstitute(
        string $entityClass,
        string $entityId
    ): EventSourceEntityInterface;
}
```

es-manager-interface.php hosted with ❤ by **GitHub** **view raw**

The manager is going to use an event source store to **store** and **restore** *events* from a *database*. It does not matter what kind of database it is going to be. Maybe MySQL, maybe PostgreSQL.

This is the beauty of interfaces — **it does not matter.**

```php
<?php

declare(strict_types=1);

namespace App\Es\Contract;

/**
 * Persistence store for event source events.
 */
interface EventSourceStoreInterface
{
    /**
     * Store ES events in a storage.
     *
     * @param class-string<EventSourceEntityInterface> $entityClass
     * @param iterable<EventInterface> $events
     */
    public function store(
        string $entityClass,
        string $entityId,
        iterable $events
    ): void;

    /**
     * Load ES events from a storage.
     *
     * @param class-string<EventSourceEntityInterface> $entityClass
     *
     * @return iterable<EventInterface>
     */
    public function restore(
        string $entityClass,
        string $entityId
    ): iterable;
}
```

es-store-interface.php hosted with ❤ by GitHub view raw

I guess we nailed it. This contract is enough to run a fully-fledged Event Source application. I have created some basic implementations:

- Default event collection based on `SplObjectStorage`,
- Default event source manager,
- Event source store based on array storage.

Unit testing

To unit test the entire thing I have created a simple aggregate root called `BankAccount`:

```php
<?php

declare(strict_types=1);

namespace Test\App\SharedKernel\Es\EntityExample;

use App\Es\Contract\EventCollectionInterface;
use App\Es\Contract\EventSourceEntityInterface;

final class BankAccount implements EventSourceEntityInterface
{
    private string $id;
    private EventCollectionInterface $events;
    private int $balance = 0;

    private function __construct(string $id, EventCollectionInterface $events)
    {
        $this->id = $id;
        $this->events = $events;
    }

    /** {@inheritDoc} */
    public static function create(
        string $identifier,
        EventCollectionInterface $collection,
        iterable $pastEvents
    ): self {
        $instance = new self($identifier, $collection);

        foreach ($pastEvents as $event) {
            switch (\get_class($event)) {
                case TransactionEvent::class:
                    $instance->applyTransaction($event);
                    break;

                default:
                    throw new \InvalidArgumentException();
            }
        }

        return $instance;
    }

    public function getEventSourceIdentifier(): string
    {
        return $this->id;
    }
    public function popEventSourceEvents(): EventCollectionInterface
    {
        return $this->events->popEvents();
    }
```

```
<?php

declare(strict_types=1);

namespace Test\App\SharedKernel\Es\EntityExample;

use App\Es\Contract\EventInterface;

/**
 * {@inheritDoc}
 *
 * @psalm-immutable
 */
final class TransactionEvent implements EventInterface
{
    /** @var int */
    private int $amount;

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

    /** {@inheritDoc} */
    public static function fromArray(array $eventData): self
    {
        return new self($eventData['amount']);
    }

    /** {@inheritDoc} */
    public function toArray(): array
    {
        return ['amount' => $this->amount];
    }

    public function getAmount(): int
    {
        return $this->amount;
    }
}
```

es-bank-account-transaction.php hosted with ❤ by GitHub view raw root and we haven't even mentioned any database yet. The code is fully portable and **independent of any framework.**

Topics left to cover in this publication

- SQL-based event store for persistent events,
- Command / Event bus support,
- View models optimized for reading (CQRS for free!)
- **Any requests? Hit me in the comments…**

The application is available on GitHub for free

https://github.com/dotcom-poland/symfony-es-example

Books worth reading

Implementing Domain-Driven Design

Implementing Domain-Driven Design [Vernon, Vaughn] on Amazon.com. *FREE* shipping on qualifying offers. Implementing…

amzn.to

Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and…

Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows…

amzn.to

PHP Symfony Laravel Php Developers Php Development