

Event Sourcing

Let's write some history

<https://joind.in/14226>

Willem-Jan Zijderveld

-  @willemjanz
-  github.com/wjzijderveld

Qandidate.com

Rotterdam

<http://labs.qandidate.com>

Broadway

Event Sourcing and CQRS library for PHP

github.com/qandidate-labs/broadway

Event Sourcing

You are throwing away data!

 **/dev/null as a Service**

Only most
recent state
gets stored

What was the **previous**
state?

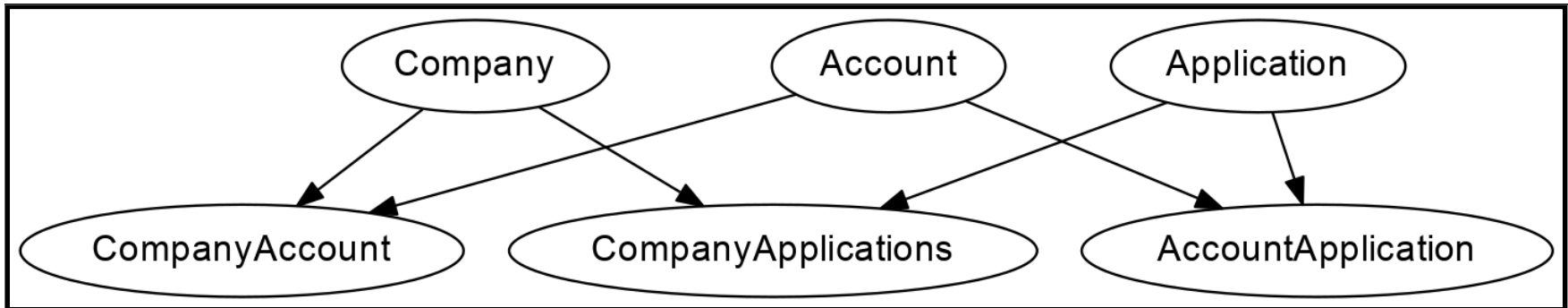
Why did the state
change?

It's hard to **keep track** of
all the data

A **company** has one or
more connected **accounts**

A company has one or more enabled applications

An **account** has access to
one or more **applications**



Complicated solutions to store and
retrieve the data

But we manage
Until...

Boss: Which accounts
received access to
application X in
December?

”We’ll add a timestamp
and we will know”

From that point in time

With CRUD you lose information

You only store the last known information

Revoking access to an application for an account

1. Log in to the account

2. Go to the account settings page

3. Find the application you want to revoke access to

4. Click on the application name

5. Click on the "Revoke Access" button

You could lose **when** the access was
revoked

You could lose **why** it got revoked

You could lose **for which application**
it was revoked

You **wouldn't know** anything anymore

Let's assume you have
a soft-delete in place

Boss: I want to know for which
applications access got revoked **more**
than once
in the last year

Keep track of revocations

account_id	app_id	datetime
75623	8	2015-01-01T00:00:00+0000
75623	8	2015-01-27T18:54:18+0000

Past revocations are gone

How does Event Sourcing
help me with that?

Store your data in
a different way

Record what has **changed**

The resulting state becomes a natural effect

EventStream

A serie of facts

CompanyRegistered

AppEnabled

AccountConnected

AccessGrantedToApp

Been there, **done** that

Single source of truth

- One source to rule all state
- The events cannot lie, it happened, deal with it

Your version control is event sourced

```
~/sandbox/git (master) $ git log
commit c1d39064d256dbe4afff8d33995d5c9e26ad7710
Author: Willem-Jan <wjzijderveld@gmail.com>
Date: Mon Jan 12 23:00:14 2015 +0100

Initial
```

```
~/sandbox/git (master) $ git reflog
c1d3906 HEAD@{0}: commit (amend): Initial
acaf468 HEAD@{1}: reset: moving to acaf4688374dd64ca32785b1a2ae3e14599ebf76
36d5893 HEAD@{2}: commit: Iteration 2
287fbe2 HEAD@{3}: commit: Iteration 1
acaf468 HEAD@{4}: commit (initial): Initial
```

Sure thing, but how?

There isn't one true answer

Event Sourcing

DDD

CQRS

All 3 are optional, but work very nice together

Domain Driven Design

The business should be reflected in your code

Entity

An model with an identity

Aggregate root

Responsible for keeping a group of entities consistent

I want to update my
model

How do I do that?

Record the change

You normally might do it like this

```
<?php // Company.php
public function __construct($id, $name)
{
    $this->id = $id;
    $this->name = $name;
}
```

But we want to record an
event

```
<?php // Company.php
public static function register($id, $name)
{
    $company = new Company();
    $company->apply(
        new CompanyRegisteredEvent($id, $name)
    );
    return $company;
}
```

Record the events

```
<php // AggregateRoot.php
public function apply($event)
{
    $this->handle($event);
    $this->uncommittedEvents[] = $event;
}
```

```
private function handle($event)
{
    $classParts = explode('\\', $event);
    $method = 'apply' . end($classParts);
    $this->$method($event);
}
```

Just one possible implementation

```
// Company.php
public function applyCompanyRegisteredEvent(
    CompanyRegisteredEvent $event
) {
    $this->companyId = $event->getCompanyId();
    $this->companyName = $event->getCompanyName();
}
```

Save your events in an
eventstore

Reloading your model

```
<php // CompanyRepository.php
function load($aggregateId)
{
    $events = $this->eventStore->load($aggregateId);

    $aggregate = new $this->aggregateClass();
    $aggregate->initializeState($events);

    return $aggregate;
}
```

```
<?php // Company.php
public function initializeState(array $events)
{
    foreach ($events as $event)
    {
        $this->handle($event);
    }
}
```

Domain Message

A message to tell your application what happened

DomainMessage

- Identifier
- Sequencenumber
- Payload
- Timestamp
- Metadata

Identifier + sequence
number

Payload

The event itself, it tells you **what**
happened

Should contain everything

It should only depend on previous events

```
final class CompanyRegisteredEvent
{
    private $companyId;
    private $companyName;

    // constructor + getters
}
```


Timestamp

It tells you **when** it happened

Metadata

Descriptive, not structural

Let's look at some other events

CompanyRegistered

AppEnabled

AccountConnected

AccessGrantedToApp

CompanyRegistered

```
// Company
function applyCompanyRegisteredEvent(
    CompanyRegisteredEvent $event
) {
    $this->companyId = $event->getCompanyId();
}
```

AccountConnected

```
// Company
function applyAccountConnectedEvent(
    AccountConnectedEvent $event
) {
    $id = $event->getAccountId();
    $this->accounts[$id] = $event->getAccountId();
}
```

AppEnabled

```
// Company
function applyAppEnabledEvent(
    AppEnabledEvent $event
) {
    $subscription = new Subscription(
        $event->getCompanyId(),
        $event->getAppId()
    );
    $this->subscriptions[$event->getAppId()] =
        $subscription;
}
```

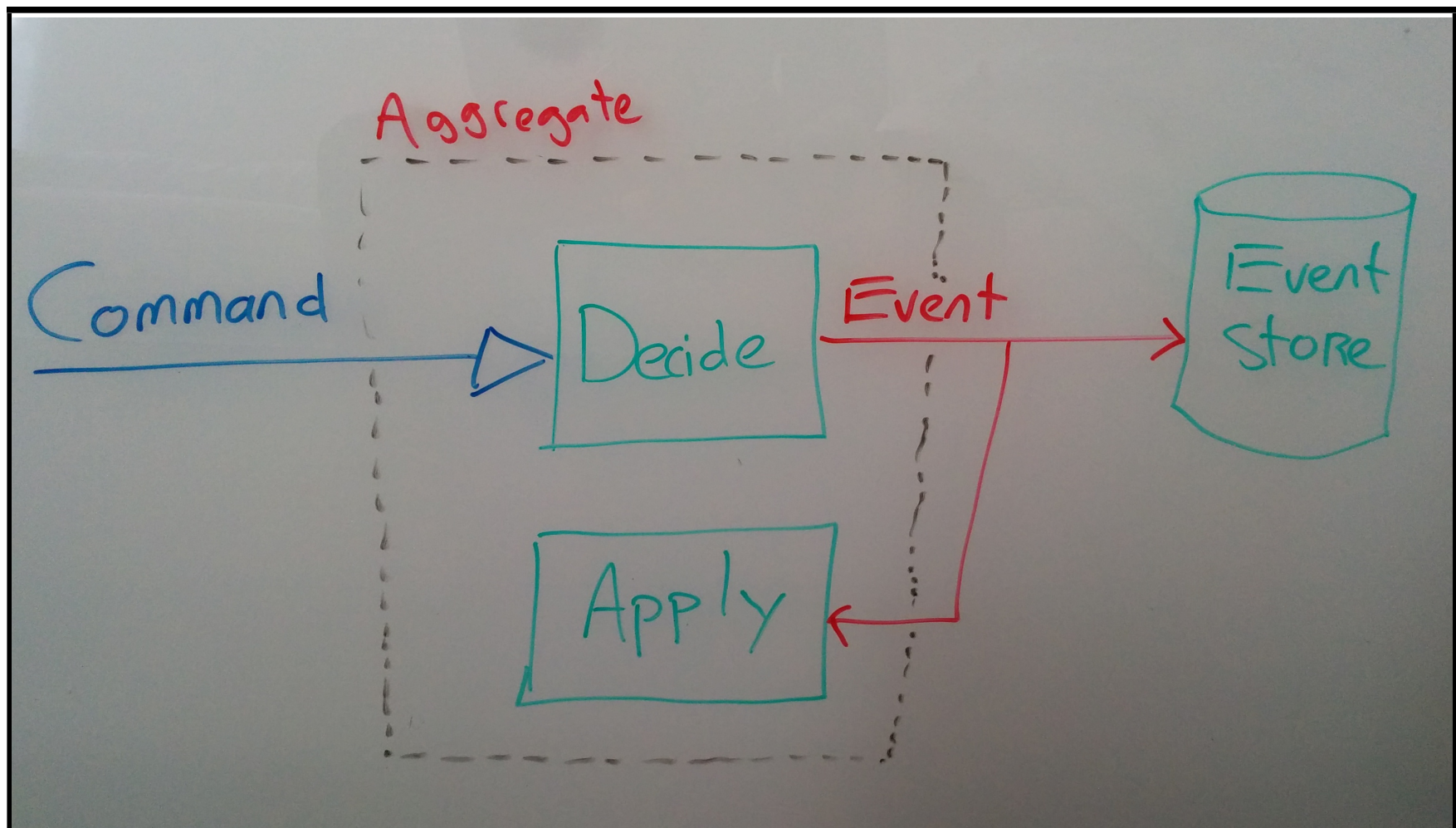
AccessGrantedToApp

```
// Company
protected function getChildEntities()
{
    return $this->subscriptions;
}
```

```
// Subscription
function applyAccessGrantedToAppEvent(
    AccessGrantedToAppEvent $event
) {
    if ($this->appId !== $event->getAppId()) {
        return;
    }

    $accountId = $event->getAccountId();
    $this->grantedAccounts[$accountId] =
        $event->getAccountId();
}
```

We can retrieve all
AccessGranted events for
December



Won't this be slow?

CQRS

Command Query Responsibility Segregation

Separate your **writes** and
your **reads**

Read Models

Create them from events using projections

Specific read models for specific views in your application

CompanyRegistration

```
class CompanyRegistration implements ReadModel
{
    public function __construct(
        $companyId,
        $companyName,
        DateTime $registeredOn
    ) {
        // ..
    }
}
```

Company Registration Projector

```
class CompanyRegistrationProjector
  implements EventListener
{
  public function applyCompanyRegisteredEvent(
    CompanyRegisteredEvent $event,
    DomainMessage $domainMessage
  ) {
    $company = new CompanyRegistration(
      $event->getCompanyId(),
      $event->getCompanyName(),
      $domainMessage->getRecordedOn()
    );
    $this->repository->save($company);
  }
}
```

Use the right tool for the
right job

Possibilities are endless

The ability to create multiple read models

- List of company registrations
- Graph of all connections between companies and accounts
- Creating reports about the amount of revocations

So.. how does this flow
work in an application?

We'll use Commands

The **write** part of CQRS

Command & CommandHandler

A CommandHandler deals with Commands and communicates with the Aggregate root

From Controller to CommandHandler

```
class CompanyController
{
    function registerAction(Request $request)
    {
        $this->commandBus->dispatch(
            new RegisterCompanyCommand(
                Uuid::uuid4(),
                $request->request->get('companyName')
            )
        );
    }
}
```

From CommandHandler to Aggregate

```
class CompanyCommandHandler
{
    function handleRegisterCompanyCommand(
        RegisterCompanyCommand $command
    ) {
        $company = Company::register(
            $command->getCompanyId(),
            $command->getCompanyName()
        );

        $this->aggregateRepository->save($company);
    }
}
```

From Aggregate to Event

```
public static function register($companyId, $name)
{
    $company = new Company();
    $company->apply(
        new CompanyRegisteredEvent($companyId, $name)
    );

    return $company;
}
```

```
public function applyCompanyRegisteredEvent(
    CompanyRegisteredEvent $event
) {
    $this->companyId = $event->getCompanyId();
}
```


From Aggregate Repository to EventStore

```
// CompanyRegistry.php
function save($aggregate)
{
    $events = $aggregate->getUncommittedEvents();
    $this->eventStore->append($aggregate->getAggregateId(), $events);
    $this->eventBus->publish($events);
}
```

From event to read model

```
class CompanyRegistrationProjector {  
    public function applyCompanyRegisteredEvent(  
        CompanyRegisteredEvent $event,  
        DomainMessage $domainMessage  
    ) {  
        $company = new CompanyRegistration(  
            $event->getCompanyId(),  
            $event->getCompanyName(),  
            $domainMessage->getRecordedOn()  
        );  
        $this->repository->save($company);  
    }  
}
```

Controller -> CommandBus -> CommandHandler ->
AggregateRoot -> Event -> EventStore -> EventBus -> Projectors
-> Read Model

Quit a bit to chew

Try not to reinvent everything yourself

Testing

Scenario based testing

Given - When - Then

Test your Command Handler

```
$this->scenario
->given([
    new CompanyRegisteredEvent(123)
])
->when(new EnableAppForCompanyCommand(42, 123))
->then([
    new AppEnabledEvent(42, 123)
]);
```

Or your aggregate

```
$this->scenario
->given([
    new CompanyRegisteredEvent(123)
])
->when(function ($company) {
    $company->enableApp(42);
})
->then([
    new AppEnabledEvent(42, 123)
]);
```

Don't forget your read models

```
$this->scenario  
  ->given([])  
  ->when(  
    new CompanyRegisteredEvent(123, 'Acme Inc')  
  )  
  ->then([  
    new CompanyRegistration(123, 'Acme Inc')  
  ]);
```


Time travel is possible!

Use events you recorded to create a new report
multiple years after the fact

You made a mistake in a projection?

So what? Correct your projector and recreate your read model
from your event stream

Questions?

[joind.in/14226](https://joinind.in/14226)

@willemjanz

Freenode: #qandidate

More information

- <http://codebetter.com/gregyoung/2010/02/20/why-use-event-sourcing/>
- <http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/>
- <http://martinfowler.com/eaDev/EventSourcing.html>
- <http://martinfowler.com/bliki/CQRS.html>
- <http://www.axonframework.org/docs/2.3/domain-modeling.html>
- <http://labs.qandidate.com/>