CQRS+ES

Command and Query Responsibility Segregation
&
Event Sourcing

enrico bono
senior full stack web dev @ SOISY
SOURCE

http://scala-phase.org/talks/duncan-devore-cqrs/
http://labs.quandidate.com
MODEL DRIVEN DESIGN

• Entities
• Value Objects
• Services
• Modules
• Aggregates and Aggregate Roots
• Bounded Contexts
• Context Maps
• And more ...
WHAT IS CQRS?

Has its origins in a concept called Command Query Separation.

- Methods should be either commands or queries.
- A query returns data, does not alter the state.
- A command changes the state, does not return data.
- Becomes clear what does and does not change state.
A STEP FURTHER

CQRS takes this principle a step further to define a simple pattern.

*CQRS is simply the creation of two objects where there was previously only one. The separation occurs based upon whether the methods are a command or a query (the same definition that is used by Meyer in *Command and Query Separation*: a command is any method that mutates state and a query is any method that returns a value).* —Greg Young
TWO DISTINCT PATHS

- One for writes (commands). One for reads (queries).
- Allows separate optimization of each of these concerns.
REASON FOR SEGREGATION

• Large imbalance between the number of reads and writes.
• Domain (command side) involves complex business logic.
• A single model encapsulating reads/writes does neither well.
• Segregation can also occur at the data store level.
  • Write side may use *something* optimized for writes.
  • Read side uses denormalized *something else* store for fast queries.
THE Q IN CQRS

• CRUD = PAIN
• DTOs are handled by projecting off the domain.
• DTOs are a different model than the domain.
• Reads from the data store ← projected date
• No longer need a complex ORM.
• Queries much easier to optimize.
• Queries much faster.
• No more looping!
THE C IN CQRS

Commands are a request of the system to perform a task or action. A sample command would be *register client x* or *change client x’s locale.*
COMMANDS

- Commands are imperative.
- They are a request to mutate state.
- They represent an action the client would like to take.
- They transfer in the form of messages or than DTOs.
- Implies a task-based UI → Command Bus
- Multiple read models or aggregates can listen to the write commands
EVENTUAL CONSISTENCY

By applying CQRS the concepts of Reads and Writes have been separated. If we keep the paths segregated, how do we keep them consistent?

• Eventual Consistency.
• Pushed asynchronously from the write side.
• Read side has listeners.
• Read side can have many listeners
• Queue can be used → Use the Event Store as your queue.
WHAT IS EVENT SOURCING (ES)?

The majority of business applications today rely on storing current state in order to process transactions. As a result in order to track history or implement audit capabilities additional coding or frameworks are required.
THIS WAS NOT ALWAYS THE CASE

- Side-effect of the adoption of RDBMS systems.
- High performance, mission critical systems do not do this.
- RDBMS’s do not do this internally!
EVENT SOURCING IS ABOUT CAPTURING EVENTS

• Its behavioral by nature.
• Tracks behavior by transactions.
• It does not maintain current state.
• Current state is derived.
CRUD SHOPPING CART

CRUD application example for processing a shopping cart order:

1. Cart created.
2. Item 1 @ $30 added.
3. Item 2 @ $15 added.
4. Item 3 @ $12 added.
5. Item 4 @ $5 added.
6. Shipping information added.
7. Total @ $62 generated.
8. Order 123 inserted.
CRUD SHOPPING CART

Now at some time in the future before the order is shipped, the customer changes their mind and wants to delete an item.

1. Order 123 fetched.
2. Item 2 @ $15 removed.
3. Total @ $47 regenerated.
4. Order 123 updated.
CRUD SHOPPING CART

Now the manager asks the development team to give him a report of all orders where customers have removed items. Since only the current state of the data is recorded, this cannot be done.

- The development team will add in a future sprint?
- Once added, it will only work from now forward.
- Substantial implications to the value of the data.
In a typical CRUD application the behavior of the system is create, read, update and delete. This is not the only way the data can be viewed.

Consider for a moment the notion of a transaction.

- Represent change between two points.
- Commonly referred to as Deltas.
- In static state models Deltas are implicit.
- They are left to frameworks such as an ORM.
- ORMs save state, calculate differences, update backing model.
- As a result much of the intent or behavior is lost.
THE CANONICAL EXAMPLE

In mature business models the notion of tracking behavior is very common. Consider for example an accounting system.

<table>
<thead>
<tr>
<th>Date</th>
<th>Comment</th>
<th>Change</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/2012</td>
<td>Deposit from $300</td>
<td>+10000.00</td>
<td>10000.00</td>
</tr>
<tr>
<td>1/3/2012</td>
<td>Check 1</td>
<td>-1000.00</td>
<td>6000.00</td>
</tr>
<tr>
<td>1/4/2012</td>
<td>ATM withdraw</td>
<td>-3.00</td>
<td>5997.00</td>
</tr>
<tr>
<td>1/11/2012</td>
<td>Check 2</td>
<td>-5.00</td>
<td>5992.00</td>
</tr>
<tr>
<td>1/12/2012</td>
<td>Deposit from $301</td>
<td>+2000.00</td>
<td>7992.00</td>
</tr>
</tbody>
</table>
THE CANONICAL EXAMPLE

• Each transaction or delta is being recorded.
• Next to it is a denormalized total of the state of the account.
• To calculate, the delta is applied to the last known value.
• The last known value can be trusted.
• Its recreated by replaying all the transactions (events).
• Its can be reconciled to ensure validity.
• The data itself is a verifiable audit log.
• The Current Balance at any point can be derived.
• State can be derived for any point in time.
EVENTS

These Deltas produced from a system built around Domain Driven Design are captured in the form of Events.

- Events are notifications.
- They report on something that has already happened.
- An event would be something like *client x has registered* or *client x locale has changed*. 
SHOPPING CART REDUX

Lets go back and take a look at Shopping Cart example and see how we manage the data from an event based perspective.

1. Cart created.
2. Item 1 @ $30 added.
3. Item 2 @ $15 added.
4. Item 3 @ $12 added.
5. Item 4 @ $5 added.
6. Shipping information added.
7. Order 123 event stream inserted.
SHOPPING CART REDUX

Now at some time in the future before the order is shipped, the customer changes their mind and wants to delete an item.

1. Order 123 event stream fetched.
2. Item 2 @ $15 removed event.
3. Order 123 event stream appended.
SHOPPING CART REDUX

This event based approach can be applied to any type of object. By replaying the event stream the object can be returned to the last known state.

- There is a structural representation of the object → It exists only by replaying previous transactions.
- Data is not persisted structural.
- It is a series of transactions.
- There is no coupling between current state in the domain and storage.
NO CRUD EXCEPT CREATE & READ

• There are no updates or deletes.
• Everything is an event.
• It's stored in append only fashion.
• Delete is simply another event that gets appended.
TECHNOLOGY IMPLICATIONS

• The storage system becomes an additive only architecture.
• Append-only architectures distribute.
• Far fewer locks to deal with.
BUSINESS IMPLICATIONS

- Criteria is tracked from inception as an event stream.
- You can answer questions form the beginning of time.
- You can answer questions not asked yet!
- Natural audit log.
SAVING OBJECTS

- CRUD applications generally require complex tools.
- Use ORM’s to track changes (Deltas).
- They keep two copies of the model to compare.
- Event based systems have no need of such complex tools.
- Each delta is captured as an event and persisted.
LOADING OBJECTS

• CRUD Applications require complex object graphs.
• Event stores != complex object graphs.
• Event stores do not require ORMs!
• One simply loads the Aggregate and replays its events.
• Current state is derived.
STRUCTURE

• Event Stores generally have very simple structures.
• Generally require only two tables.
• One for the Aggregates.

• Events table
  • uuid
  • version
  • type
  • payload
  • metadata
REPLAY

• Found a bug at day one? Replay all the data!
• Added a property to cart object? Replay all the data!
• Wanna know how may products were removed from the cart? Replay the data!
SAMPLE IMPLEMENTATION: BROADWAY BY QANDIDATE

- Labs.Qandidate.com
COMMAND BUS

```php
$createInvestor = new CreateInvestor(
    $userId,
    $fiscalCode,
    $request->get('gender'),
    $request->get('birthday'),
    new \DateTime()
);

$this->dispatch($createInvestor);
```
public function handleCreateInvestor(CreateInvestor $command)
{
    User::createInvestor(
        $command->getUserId(),
        $command->getFiscalCode(),
        $command->getGender(),
        $command->getBirthday(),
        $command->getDate()
    );
}

public function handleRegisterUser(RegisterUser $command)
{
    /** @var User $user */
    $user = $this->repository->load($command->getUserId());

    $user->registerUser(
        $command->getFirstname(),
        $command->getLastname(),
        $command->getEmail(),
        $this->passwordEncoder->encodePassword($command->getUser(), $command->getPassword()),
    );

    $this->repository->save($user);
}

public function handleConfirmUser(ConfirmUser $command)
{
    $user = $this->repository->load($command->getUserId());

    $user->confirmUser(
        $command->getConfirmationToken(),
        $command->getConfirmedAt()
    );

    $this->repository->save($user);
}
public function registerUser($firstname, $lastname, $email, $password, $confirmationToken)
{
    if ($this->state == self::REGISTERED || $this->state == self::CONFIRMED) {
        return;
    }

    $this->apply(new UserWasRegistered($this->userId, $this->fiscalCode, mb_convert_case($firstname, MB_CASE_TITLE, 'UTF-8'), mb_convert_case($lastname, MB_CASE_TITLE, 'UTF-8'), $email, $password, $confirmationToken));
}
protected function applyUserWasRegistered(UserWasRegistered $event)
{
    /** @var User $readModel */
    $readModel = $this->repository->find($event->getUserId());
    $readModel->setIsRegistered();
    $readModel->setConfirmationToken($event->getConfirmationToken());
    $readModel->setFirstname(mb_convert_case($event->getFirstname(), MB_CASE_TITLE, 'UTF-8'));
    $readModel->setLastname(mb_convert_case($event->getLastname(), MB_CASE_TITLE, 'UTF-8'));
    $readModel->setEmail($event->getEmail());
    $readModel->setPassword($event->getPassword());
    $this->repository->save($readModel);
USE CASE

```php
public function loadUserByUsername($username) {
    $results = $this->userRepository->findBy(['email' => $username]);

    if (count($results) == 1) {
        /** @var \Soisy\Domain\IdentityAccess\ReadModel\User $user */
        $user = $results[0];
    }
```
$this->scenario
  ->given()
    ->new InvestmentCancellationRequested()
      ([$investment->getInvestmentId(),
        $investment->getInvestmentAccountId(),
        $investorEmail,
        $investorPaymentAccountId,
        $totalInvestedAmount,
        $removedAt])
  
  ->when()
    ->new InvestmentRemoved()
      ([$investment->getInvestmentId(),
        $investment->getInvestmentAccountId(),
        $investorEmail,
        $investorPaymentAccountId,
        $totalInvestedAmount,
        $removedAt])
  
  ->then()
    ->new DirectTransfer()
      ([$new PaymentTransactionId($transactionId),
        'PUBLICID',
        'PUBLICID2',
        $investment->getInvestmentAccountId(),
        $investorEmail,
        $investorPaymentAccountId,
        'investor@example.com',
        $totalInvestedAmount,
        CausalFactory::create(RefundForInvestmentCancellation::CAUSAL_INVESTMENT_REMOVED_ID)
        $removedAt,
        CausalFactory::create(RefundForInvestmentCancellation::CAUSAL_INVESTMENT_REMOVED_ID)])
CONCLUSION

When combined together CQRS and Event Sourcing provide a very powerful platform for complex domains.

- Structured separation of read and write concerns.
- Allowing distinct read / write optimization.
- Explicitly it represents a behavioral based architecture.
- More clearly represents a Domain Driven Design approach.
- It demands a complete Event Log.
- Has a derived state model rather than a static state model.
THAT'S ALL, FOLKS!