The Fine Art of Time Travelling: implementing Event Sourcing

Andrea Saltarello

Software Architect @ **managed/designs**

https://twitter.com/andysal74

andysal@gmail.com

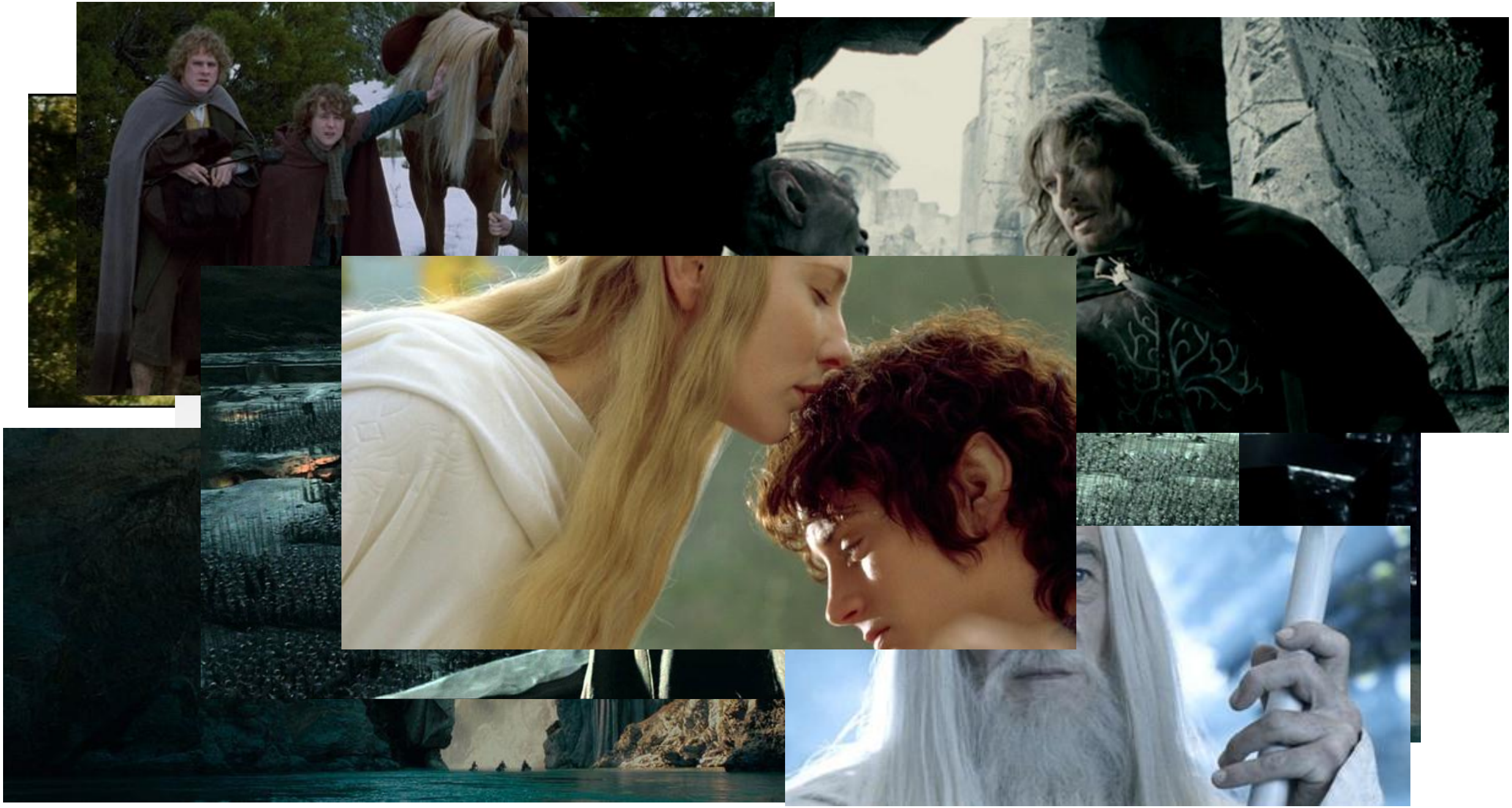
There’s no silver bullet

Not the way to implement Event Sourcing, but a working way to do it nonetheless

Demo app available on Github (GPL3), BTW

The (ambiguous) Lord of the Rings

A few fancy dressed blokes making a jaunt



andysal74

It really became clear to me in the last couple of years that we need a new building block and that is the Domain Event.

[Eric Evans]

An event is something that has happened in the past.

[Greg Young]

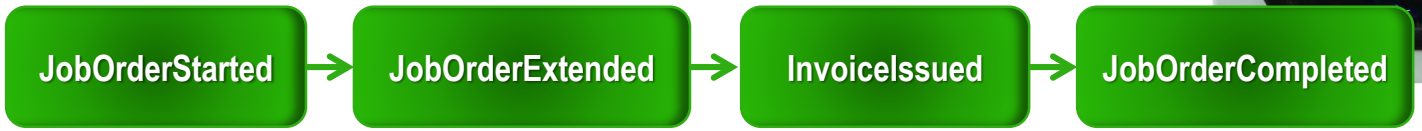
A domain event … captures the memory of something interesting which affects the domain

[Martin Fowler]

Event Sourcing in a nutshell

Instead of focusing on a system's last known state, we might note down every occurring event: this way, we would be able to (re)build the state the system was in at any point in time just replaying those events

To cut a long story short: we'd end up recording an **event stream**



What's an event, anyway?

The (immutable) composition of:

- A (meaningful) name
- (Typed) Attributes

ProjectCompleted
DateOfCompletion
ProjectId

InvoiceIssued
DateOfIssue
Customer
Price

ProjectStarted
DateOfStart
ProjectId

ProjectRegistered
DateOfRegistration
DateOfEstimatedCompletion
ProjectId
CustomerId
Price

demo

Event Stream

Events vs. Relations

Although replaying a DBMS event log or adopting a temporal database would allow to restore a specific system state, we would miss the reason behind every occurred change nonetheless

```
INSERT INTO X (M, L, G) VALUES (1, 0, 1)
UPDATE X SET M=X, L=Y … WHERE …
UPDATE X SET M=42 … WHERE …
UPDATE X SET J=K … WHERE …
```

```
INSERT JobOrderStarted VALUES ()
INSERT JobOrderExtended VALUES ()
INSERT InvoiceIssued VALUES ()
INSERT JobOrderCompleted VALUES ()
```

Event Stream vs. «My application»

Still, my users are more interested in knowing a job order's balance or whether an invoice has been paid. (cit.)

That is, we need a way to produce an entity state

Event Sourcing <3 DDD

DDD's Aggregates provide a convenient way to encapsulate event management

***Aggregate**: A collection of objects that are bound together by a root entity, otherwise known as an aggregate root. The aggregate root guarantees the consistency of changes being made within the aggregate.*

[Wikipedia]

An aggregate is responsible for:

- encapsulating business logic pertaining to an "entity"
- generating events to have them available for saving
- replaying events in order to rebuild a specific state

demo

Aggregates

Aggregates vs. Events vs. Repos

```
var aggr = repository.GetById<TAggr>(id);    //Triggers [time travelling capable] event replay
aggr.DoSomething();                          //Domain logic + events raising
repository.Save(aggr);                       //Updates the event stream + events dispatching
```

Repository: *Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.* [DDD]

demo

Time Travelling

Event Stream vs. «My application»

Still², my users are more interested in knowing a job order's balance or whether an invoice has been paid. Quickly.

Ways to achieve that:

- Snapshots can help
- CQRS to the rescue: let's have a database storing the usual «last known system state» using it as a *read model*

demo

Snapshots

Enter CQRS

Acronym for **C**ommand **Q**uery **R**esponsibility **S**egregation

Basically, ad hoc application stacks for both writing and reading:

- “Command” stack writes events and snapshots
- “Read” stack reads from eventually consistent, reading purposes optimized database(s)

CQRS: the “Read” side of the Force

As a business unit manager, I want to collect credits due to unpaid outgoing invoices #ubiquitouslanguage #nuffsaid

CQRS/ES wise, this user story could be implemented by means of the following real world C# code:

```
Database.OutgoingInvoices.
    .PerBusinessUnit(businessUnitId)
    .ExpiredOnly()
    .Select(i => new {InvoiceNumber = i.Number, CustomerId = i.Customer.Id})
    .AsParallel()
    .ForAll(i => bus.Send(new CollectDebtCommand(i.InvoiceNumber, i.CustomerId)));
```

demo

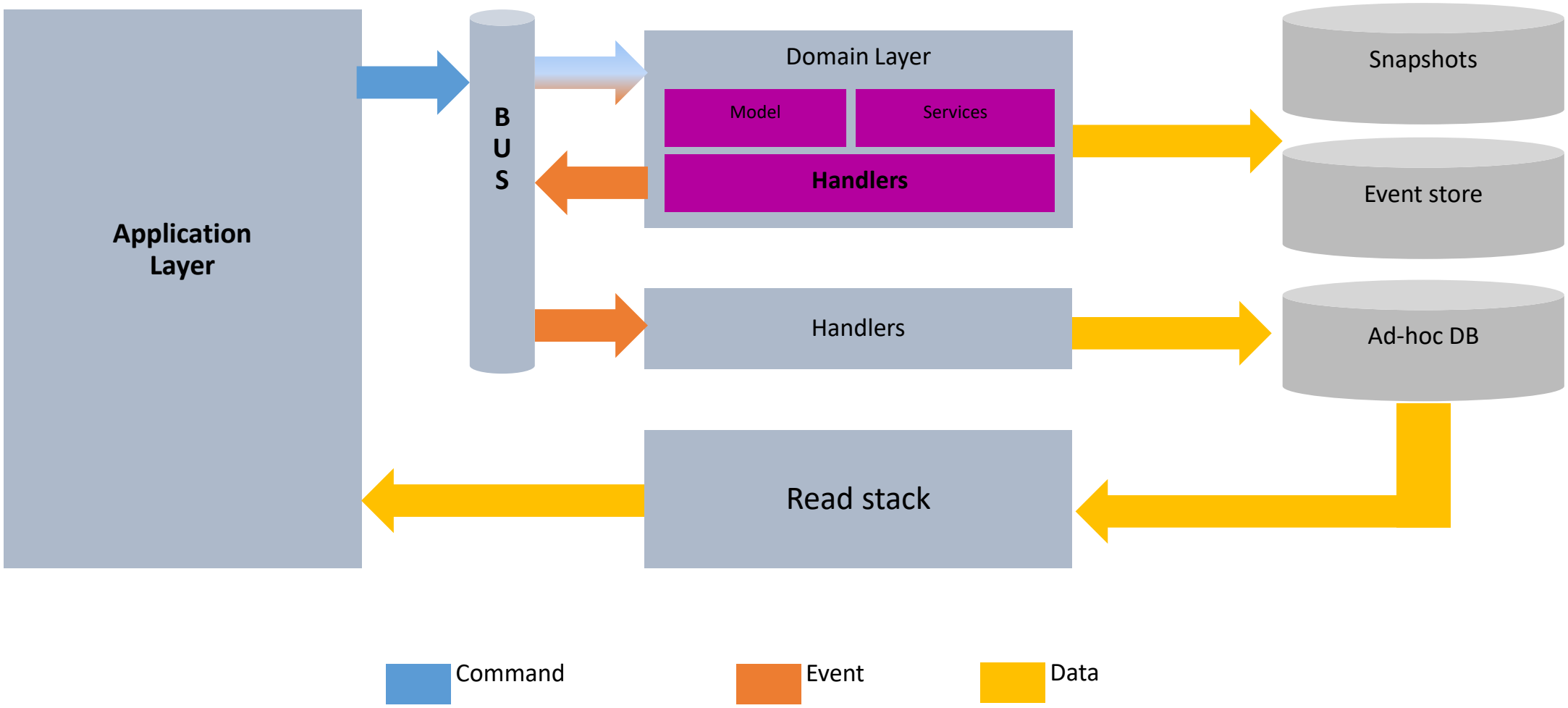
Read Model

CQRS/ES in a nutshell

1. Application sends a command to the system
2. Command execution might alter the system's state and then raise events to state success/failure
3. Events are notified to interested subscribers (a.k.a. handlers), such as:
    - Workflow managers (a.k.a. «Sagas») which could execute more commands
    - Denormalizers, which will update the read model database

Note: command/event dispatch/execution will usually be managed by a Mediator («bus»)

CQRS/ES at a glance



demo

Handlers/Denormalizers

Oh, just one more thing ☺

Time Travelling… «Fringe» style

What if we could manage events occurring in parallel, alternative timelines?

demo

Speculations

THANK YOU!

Encore

A bus! A bus! My kingdom for a bus!

Although building a simple bus might be feasible (and tempting as well ☺), 3°-party products (e.g.: MassTransit, NServiceBus, Rebus, …) provide much-needed features such as:

- Fault tolerance
- Scalability
- Workflow support
- Events' scheduling

demo

Bye, bye, polling (Polling Goodbye)