Introduzione a CQRS ed Event Sourcing

Dev Romagna

Chi siamo

Riccardo Franconi

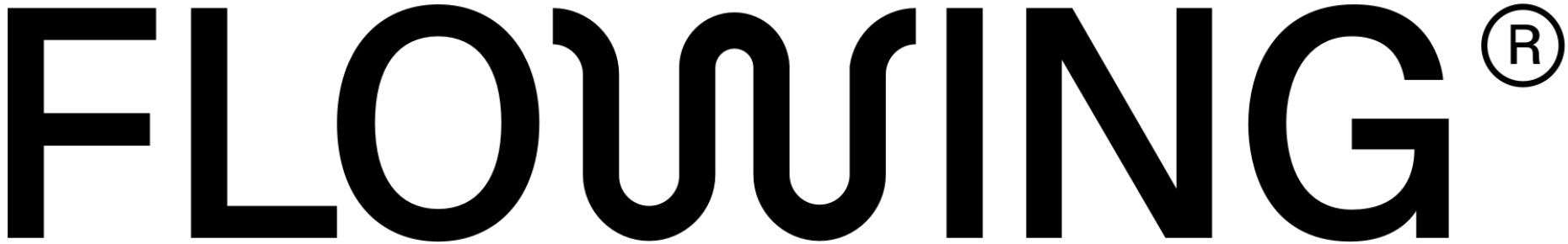
@ricfrank

babbo, sviluppatore, musicista

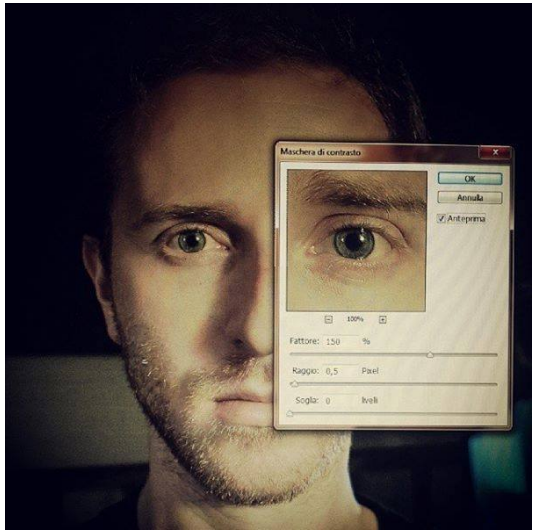
SOISY

Il Marketplace per il pagamento rateale su e-commerce

www.soisy.it



www.flowing.it



Matteo Galacci

Back-end dev freelance e consulente

Github: matiux
Slack GrUSP: Matteo Galacci
Linkedin: Matteo Galacci

Cosa vedremo

- CQRS/ES
- Alcuni concetti del DDD utili alla comprensione di CQRS/ES
- Demo

Cos’è CQRS

ASPETTATIVE

Non diventerete esperti di CQRS/ES e DDD ma cercheremo di introdurvi a questo mondo, lasciandovi del codice di esempio per evitare di partire da zero

Cos’è CQRS

Cos’è CQRS ?

Cos’è CQRS

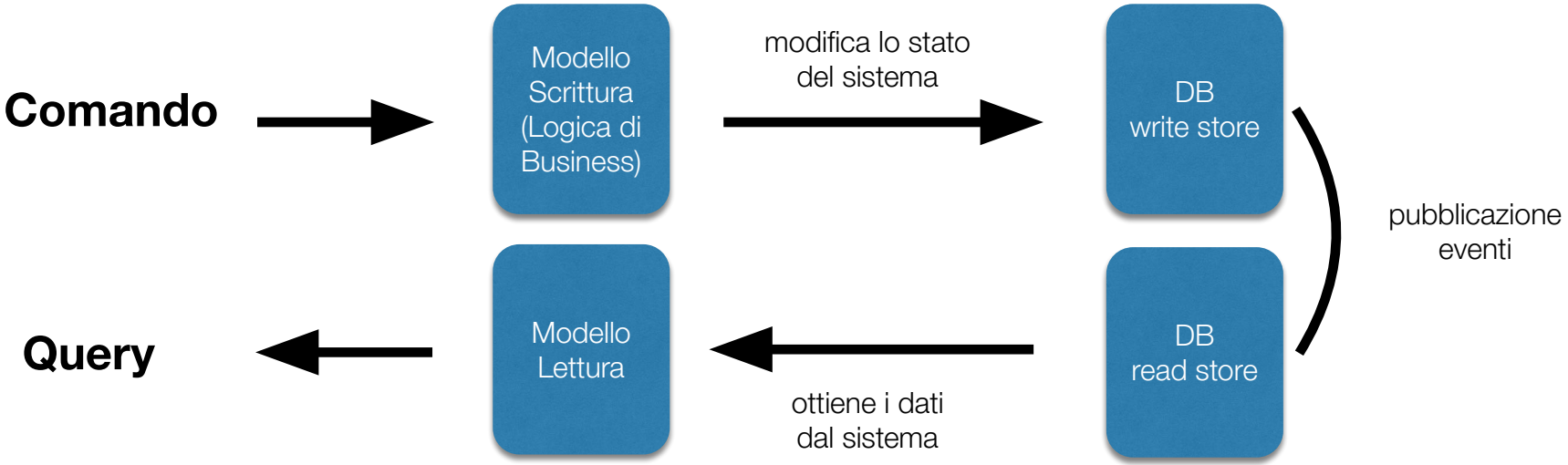
CQRS (Command/Query Responsibility Segregation) è un architettura utilizzata per la **modellazione del dominio di un software**.

Ogni azione che viene eseguita in un software può essere definita come **comando** o come **query**: il comando è un’operazione che modifica lo stato del sistema, la query è un’operazione di lettura che serve a reperire delle informazioni lasciando il sistema inalterato.

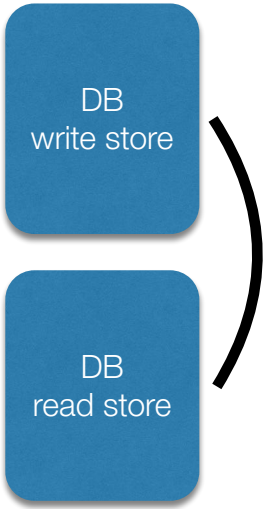
Cos’è CQRS

Possiamo utilizzare un **modello differente** per operazioni di scrittura (**command**), rispetto a quello utilizzato per la lettura (**query**)

Comando, Query e modelli separati



Comando, Query e modelli separati



La comunicazione tra write side e read side avviene
tramite la pubblicazione di uno stream di eventi (uno o più eventi)
che vengono poi salvati nella stessa transazione

Si può utilizzare anche lo stesso DB

Modelli separati scrittura/lettura - Vantaggi

- Modelli più semplici, ortogonali e coesi
- Query in scrittura/lettura più semplici e performanti (il dato è costruito ad-hoc per la vista)
- Flessibilità in risposta a cambiamenti (interfacce sempre più complesse, frequenti evoluzione delle logiche di dominio)

CQRS vs. CRUD

- CRUD difficile da gestire quando la rappresentazione del dato per la **scrittura è incompatibile con quella per la lettura**. Ci troviamo a fare modifiche per la lettura anche quando non è necessario per la scrittura.

- **Operazioni di lettura sempre più complicate** (query difficili da gestire ad esempio per interfacce complesse).

CQRS vs. CRUD

- Difficoltà di scalabilità in termini di **performance** (query in lettura con molte join).
- Mette il **focus sul dato e non sul comportamento**.

CQRS vs. CRUD

- No data-centric ma **behavior-centric.**
- I modelli (commands, entities, vo, aggregates, read models) **parlano il linguaggio di dominio** (no dominio anemico).

Quando usare CQRS

- **Domini complessi (es: flussi complessi) o porzioni di essi.**
- E’ utile gestire **performance di scrittura e lettura separatamente.** In molti sistemi il numero di letture è nettamente superiore a quello di scritture.

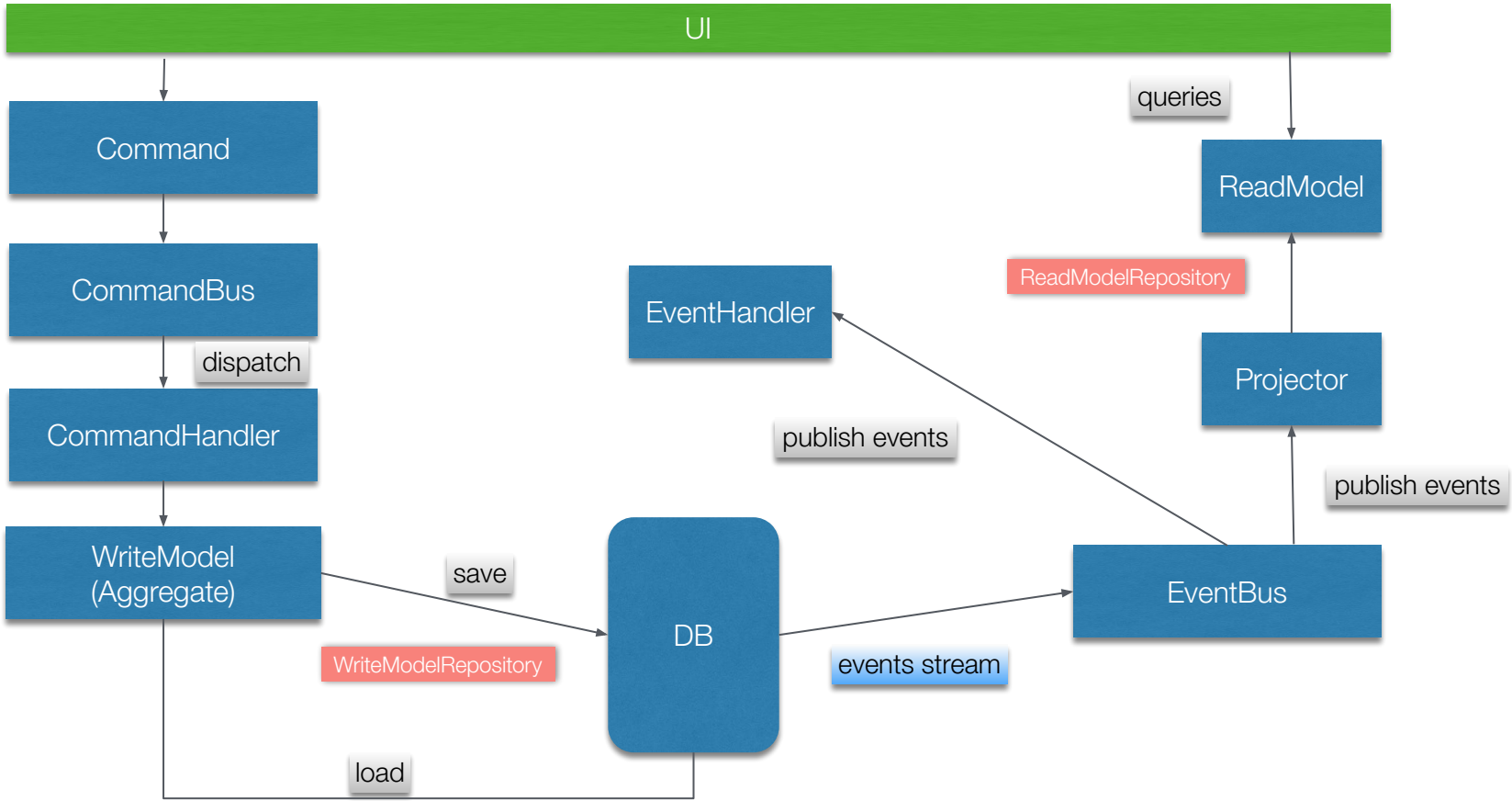
Quando usare CQRS

- Ci sono dei **team** che possono lavorare solo su parti del dominio complesse (scrittura) e altri su parti più semplici (lettura)
- **Regole di business cambiano frequentemente**. C'è bisogno di flessibilità e di mantenere la complessità del software lineare.
- Interfacce complesse

Quando non usare CQRS

- **Domini semplici** per cui CRUD è sufficiente**.**
- **Per implementazioni sull'intero sistema**. Va usato solo in parti dove è realmente utile. Evitiamo di aggiungere complessità non necessaria.
- Non possiamo usare l'**eventual consistency**.

CQRS


UI
Command
CommandBus
dispatch
CommandHandler
WriteModel
(Aggregate)
save
WriteModelRepository
load
DB
queries
ReadModel
ReadModelRepository
EventHandler
Projector
publish events
publish events
EventBus
events stream

Cos’è Event Sourcing ?

Cos’è Event Sourcing

Pattern che **utilizza gli eventi come strumento per modellare il dominio**.

Lo stato dell’applicazione e i suoi cambiamenti vengono rappresentati come una **sequenza di eventi** salvati nella base di dati, chiamata **event store**.

Event Store

Id	Event
1	UserRegistered
2	ItemAddedToCart
3	ItemAddedToCart
4	OrderPlaced
5	PaymentDone
6	…

Cos’è ES

Perché ?

- Non è più sufficiente tenere traccia dell'ultimo stato del sistema
- Quando vogliamo **catturare l'intento**, il motivo o lo scopo per cui un dato è stato cambiato da un utente (Account opened, Account closed, ecc)

Cos'è ES

Evento

è **un’azione passata**, come ad esempio “prestito approvato”

Evento

```php
class UserRegistered
{
    private $userId;
    /**
     * @var DateTime
     */
    private $occurredAt;


    public function __construct($userId, \DateTimeImmutable $occurredAt)
    {
        $this->userId = $userId;
        $this->occurredAt = $occurredAt;
    }

    /**
     * @return mixed
     */
    public function getUserId()
    {
        return $this->userId;
    }
}
```

Cos’è ES

Modello come sequenza di eventi

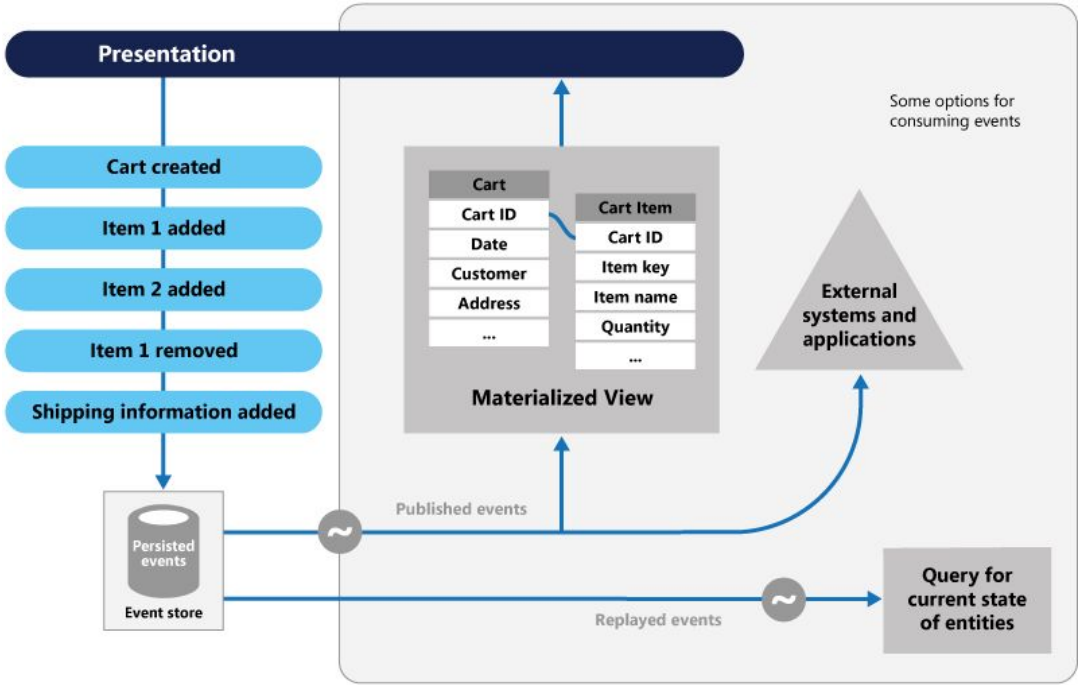
- Rappresenta i modelli come **una sequenza di eventi** che si è verificata nel tempo e che li ha portati ad essere nel loro stato attuale.
- Tutti i cambiamenti di stato dell'applicazione vengono quindi conservati ed è sempre possibile recuperarli e **ricostruire la storia passata**.


ItemAdded
ItemAdded
ItemRemoved
ItemsOrdered
OrderPaid
OrderShipped

Cos’è ES

Append only

- **Non esistono operazioni di modifica o cancellazione di eventi**, possiamo solo aggiungere gli eventi che succedono.
- Dopo aver salvato l'evento, di solito l'**event store pubblica l'evento** che potrà essere consumato da chi sta in ascolto (pub/sub pattern).


Presentation
Cart created
Item 1 added
Item 2 added
Item 1 removed
Shipping information added
Persisted events
Event store
Some options for consuming events
Cart
Cart ID
Date
Customer
Address
...
Cart Item
Cart ID
Item key
Item name
Quantity
...
Materialized View
External systems and applications
Published events
Replayed events
Query for current state of entities

Cos’è ES

Persistenza

Si ha un’unica tabella che viene chiamata **Event Store** che conterrà **tutti gli eventi** che vengono salvati durante l’utilizzo del software

Persistenza

Aggregate	Version	Event	Details
OLJ 565 M	1	*Purchased*	*Cost = 1700, Vendor = JWG Cars Ltd*
OLJ 565 M	2	*Resprayed*	*Colour = Mallard Green*
OLJ 565 M	3	*Insured*	*Insurer = AIG, Cover = Third Party*
ZV 1618	1	*Purchased*	*Cost = 3700, Vendor = Azure Autos, Payment = Cash*
OLJ 565 M	4	*Exported*	*Destination = Luxembourg*
OLJ 565 M	5	*Sold*	*Price = 2700, Purchasor = M. Chorizo*
ZV 1618	2	*Repaired*	*Detail = Sheared trunion, Parts = 200, Labour = 50*
ZV 1618	3	*Tuned*	*Mechanic = J. Smythe, Machine = 109092*
ZV 1618	4	*Insured*	*Insurer = Churchill*

Cos’è CQRS

Time travel

Siamo sempre in grado di **risalire allo stato del sistema in un determinato momento**.

Cos’è
ES

Replay eventi

Possiamo sempre **ri-proiettare gli eventi**, per costruire nuove proiezioni di dati o modificare le vecchie.

Immutabilità

Gli eventi sono **immutabili**

Quali eventi vanno salvati ?

Eventi di dominio

Hanno un significato per gli esperti di dominio

Unica sorgente di verità

- L’event store è l’unica **sorgente di verità**.
- Gli eventi non andrebbero mai modificati ma compensati con altri eventi.

Quando usare Event Sourcing

E’ importante **tenere traccia della storia** dello stato nel nostro sistema

Cos’è ES

CQRS + Event Sourcing

Perché sono due pattern che vanno molto bene insieme ?

Problema Event Sourcing

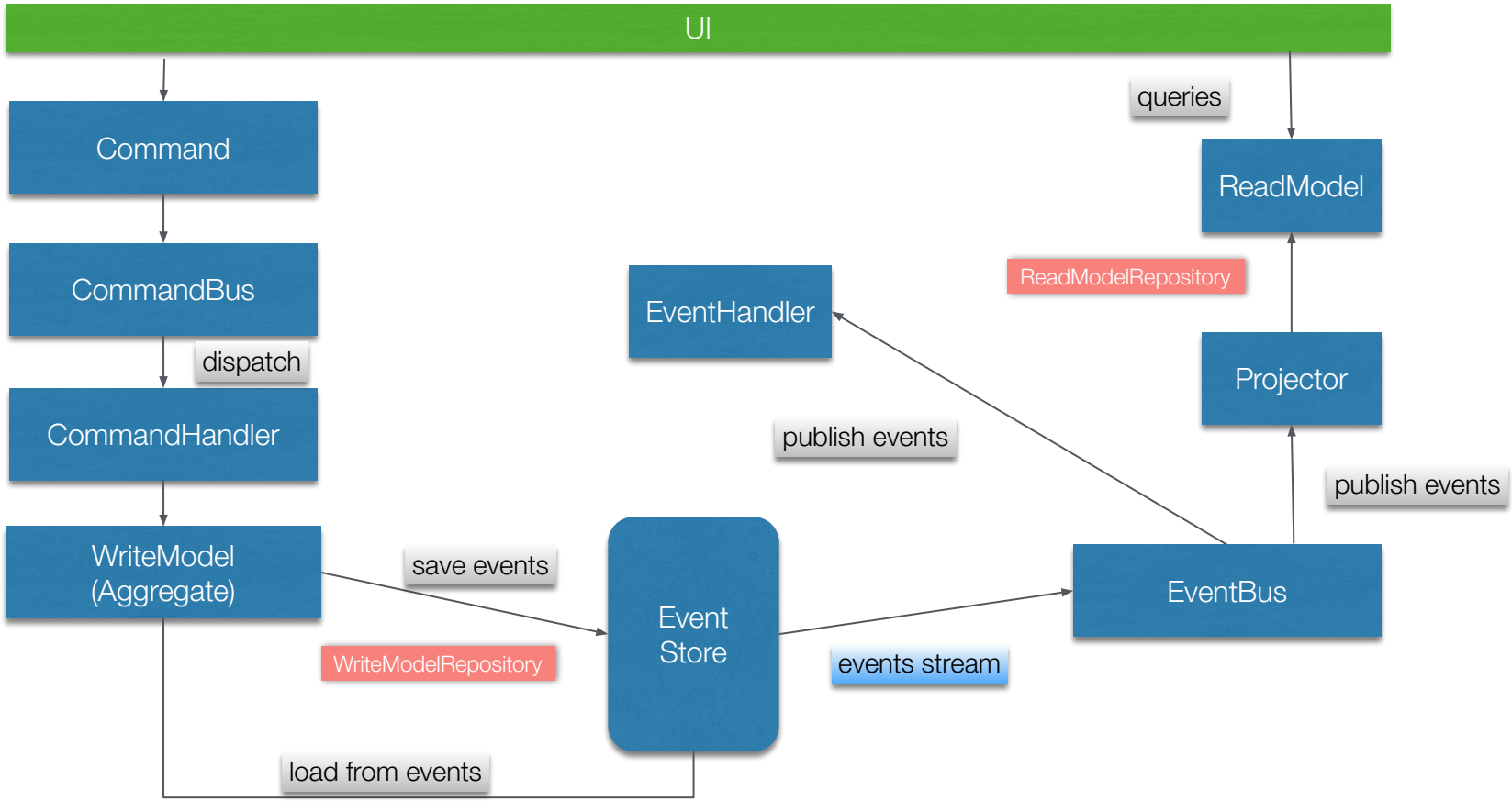
- **E’ difficile e limitante fare query sull’event store** (es: tutti gli utenti che si chiamano ‘Riccardo’)
- Con CQRS possiamo evitare di fare query specifiche direttamente sull’event store, grazie ai read model

Problema CQRS

- In CQRS, per creare dei read model, di solito si pubblicano degli **eventi che vengono salvati transazionalmente,** principalmente per motivi di replay.

- Con Event Sourcing possiamo **salvare questi eventi e utilizzarli per la costruzione dei nostri modelli.**

CQRS + Event Sourcing



Comando

- **Azione che viene eseguita sul sistema**: in OOP può essere rappresentato come un oggetto il cui nome descrive l'azione e il suo stato rappresenta tutta l'informazione necessaria a compiere quell'azione.

- E' il messaggio inviato all'aggregato e contiene i dati necessari ad eseguire la logica di business scatenata dall'azione.

Comando

```
class CreateOrder
{
    private $id;

    private $items;

    ...
}
```

Evento

- **E’ una cosa avvenuta nel nostro sistema**: in OOP può essere rappresentato come un oggetto immutabile il cui nome descrive l’azione avvenuta e il suo stato rappresenta il nuovo stato persistito (es: OrderCreated)

Evento

```
class OrderCreated
{
    private $id;

    private $items;

    ...
}
```

Bus

- **Command bus**: ha la responsabilità di inviare **comandi** agli handler registrati.
- **Event bus**: ha la responsabilità di pubblicare **eventi** agli handler registrati.

Command Handler

- Riceve il comando
- Tramite il repository dell’aggregato lo ricostruisce portandolo allo stato corrente
- Passa il comando all’aggregato che esegue la logica di business e genera uno o più eventi
- Tramite il repository dell’aggregato lo salva

Read model

- Il read model ci permette di avere una **rappresentazione “denormalizzata” del dato**, che semplifica tantissimo la logica della view e delle query di lettura

- Questo oggetto **rappresenta il dato da mostrare** ed espone solo comportamenti utili alla presentazione

Projector

- Sta in **ascolto di eventi** con i quali create/modificare/cancellare readmodel.
- Ogni projector deve poter essere in grado di ricostruire i readmodel in **maniera isolata**.

Aggregato

Grafo di oggetti di dominio che viene trattato come **una singola unità**

(un ordine con tutte i suoi dettagli)

Aggregato

Da Implementing Domain-Driven Design di Vaughn Vernon:

“Gli aggregati sono confini di consistenza accuratamente elaborati che raggruppano entità e oggetti valore.”

Aggregato

Martin Fowler:

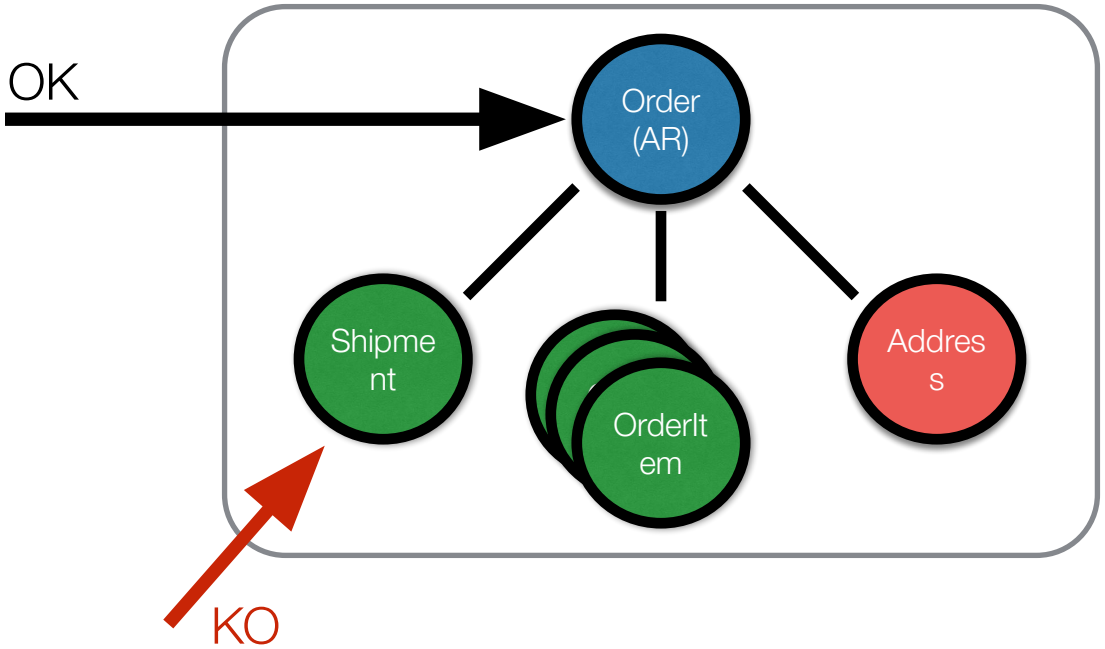
In DDD, un aggregato è una raccolta di oggetti correlati che desideriamo trattare come un'unità. In particolare, è un'unità per la manipolazione dei dati e la gestione della coerenza al fine di modificare lo stato degli aggregati con operazioni atomiche.

Aggregato

Martin Fowler:

Un aggregato è un pattern nel DDD. È un cluster di oggetti di dominio che possono essere trattati come una singola unità. Un esempio può essere un ordine e i suoi elementi, questi saranno oggetti separati, ma è utile trattare l'ordine (insieme ai suoi elementi) come un singolo aggregato. Un aggregato avrà uno dei suoi oggetti come **Aggregate Root**. Qualsiasi riferimento dall'esterno dell'aggregato passerà per la radice. La radice può quindi garantire l'integrità dell'aggregato. Gli aggregati sono l'elemento base del trasferimento quando si parla di archiviazione dei dati: si richiede di caricare o salvare l'intero aggregato. L'aggregato DDD non va confuso con un oggetto "collezione" (lists, map, ecc.); gli aggregati DDD sono concetti di dominio (ordine, visita clinica, playlist), mentre le collezioni sono generiche.

Aggregato Order



Aggregato

Transactional boundary

Tutti gli oggetti verranno salvati come **una singola unità transazionale,** garantendo la consistenza del dato e il rispetto delle **invarianti**

Aggregato

Invariante

Regola di business che deve essere **sempre vera** e **transazionalmente consistente** durante l'esecuzione dell'applicazione

(la somma del prezzo degli item deve essere uguale al prezzo totale dell'ordine)

(non posso fare il checkout di un ordine se l'ordine è stato chiuso)

Aggregato

- Definisce **logiche di business.**
- Ogni aggregate ha un **identificatore univoco**.
- L’unità singola di persistenza.

Aggregate Root (Il muro)

L'aggregato root **garantisce la consistenza** dei cambiamenti compiuti all'interno dell'aggregato proibendo ad oggetti esterni di detenere i riferimenti ai suoi membri interni

Aggregato

```php
$order = ... // order entity
$orderItem = new OrderItem('DDD in PHP', 24.99);
$order->addItem($orderItem);
$total = 0;
foreach ($order->getItems() as $item) {
    $total += $item->getPrice();
}
$order->setTotal($total);
```

Approccio classico in cui viene creato l'ordine e poi relazionato ai suoi dettagli e le singole entità vengono salvate

Aggregato

Tell don’t ask

Tutte le operazioni sono **delegate all’aggregato** e non viene esposta all’esterno la conoscenza del dettaglio dell’ordine.

Tell don’t ask ci ricorda che l'orientamento dell’oggetto riguarda il raggruppamento dei dati con le funzioni che operano su quei dati. Ci ricorda che piuttosto che chiedere a un oggetto i dati e agire su quei dati, dovremmo invece dire a un oggetto cosa fare. Questo incoraggia a spostare il comportamento all'interno degli oggetti di dominio (aggregati).

```
$order = new Order(...);
$orderLine = $order->addOrderLine('DDD in PHP', 24.99);
$orderRepository->save($order);
$orderLineRepository->save($orderLine);
```

Aggregato

Law of Demeter (LoD)

Noto anche come “**principle of least knowledge**”. Tale principio di basa su 2 punti:

1. Ogni blocco di codice (metodo, funzione ecc) deve conoscere solo pochi altre blocchi strettamente correlati;
2. Ogni blocco di codice dovrebbe interagire solo con i blocchi che conosce direttamente.

In sintesi ci dice di non parlare con gli sconosciuti.

```
$order = new Order(...);
$order->addOrderLine('DDD in PHP', 24.99);
$orderRepository->save($order);
```

Aggregato

Ubiquitous language

È il linguaggio comune e rigoroso tra gli sviluppatori, gli utenti e gli esperti di dominio. A questo è associato uno dei dubbi che spesso gli sviluppatori hanno; **codice italiano o codice inglese?** Per quanto agli sviluppatori possa piacere scrivere i software in inglese, il codice dovrebbe rispettare il più possibile il linguaggio di dominio.

Questo comporta alcuni vantaggi:

1. Il codice è auto esplicativo e quindi la documentazione è il codice stesso.
2. Quando parliamo con i colleghi di specifiche e concetti, questi hanno un rapporto 1:1 con il codice.
3. Un nuovo membro del team trarrà vantaggio dal leggere il codice e trovarsi i concetti scritti come gli sono stati spiegati.

Aggregato

Aggregato in CQRS+ES

- Questo oggetto riceve i dati da un comando, esegue logica di business e genera eventi che verranno poi salvati sull'event store rispettando le invarianti.

- Lo stato di un aggregato viene sempre ricostruito dallo **stream dei suoi eventi.**

Aggregato in CQRS+ES

- Se gli eventi sono tanti potrebbero esserci **problemi di performance** nel momento in cui il modello viene idratato
- Snapshot o rivedere il design

Transazionalità

- Se siamo su un db relazionale dobbiamo assicurarci che le varie query vengano eseguite in maniera atomica (transazioni).
- Se siamo su un NoSql salveremo un unico documento che contiene ordine e dettagli.
- Le transazioni non dovrebbero attraversare i confini dell'aggregato.

Aggregato

Vantaggi

- è un pattern che pur se complesso permette di **mantenere una complessità lineare** all'interno del software.

- l'approccio ad eventi permette di **modellare molto bene i flussi** che avvengono all'interno del dominio.

- **append only**: Per ogni azione richiesta al sistema, ci saranno nuovi eventi salvati. Questo riduce la complessità in fase di scrittura in quanto concetti come "modifica" o "cancellazione" vanno sempre gestiti nella stessa maniera ovvero salvando eventi che rappresentano una modifica o una cancellazione.

Vantaggi

- **ricostruire ogni volta che vogliamo tutti i read model**, ri-proiettando tutto l'event stream. L'event store è l'**unica fonte di verità** e proprio da questa fonte possiamo attingere per ricostruire read model specifici per le nostre esigenze.

- **l'event store è a sua volta un log**. Questo rende più semplice ricostruire cosa è accaduto nel sistema, ad esempio per avere delle **statistiche su un utente (BI)** o per il debug. Abbiamo a disposizione tutto quello che è successo nel dominio del software e possiamo andare **avanti e indietro nel tempo** (requisito che in un dominio bancario è richiesto) in base alle nostre esigenze.

Vantaggi

- **Task based UI.**
- Codice aderente al **linguaggio di dominio.**
- **Scalare in maniera indipendente** la parte di scrittura e quella di letture.

Vantaggi

- **l’evento diventa il contratto da rispettare tra scrittura e lettura.** Questo potrebbe permettere a persone differenti di poter implementare esclusivamente la parte di lettura o quella di scrittura.

- **può essere applicato solo per parti complesse**. Nessuno ci obbliga ad applicare il pattern in tutte le parti del software.

- **Event storming** come modello di discovery.

- E’ importante minimizzare o evitare conflitti su aggiornamenti di dati simultanei (append only, optimistic concurrency)

Svantaggi

- **Curva apprendimento lunga.**
- E’ un **pattern non molto comune** ed ancora non molto utilizzato dalla comunità **PHP**.
- Diventa **svantaggioso per domini semplici** come ad esempio applicazioni principalmente basati su CRUD o flussi semplici

Svantaggi

- E’ un pattern **abbastanza verboso**.
- Per certe applicazioni si potrebbe **arrivare ad avere aggregati che con moltissimi eventi** che potrebbero deteriorare le prestazioni (attenzione a come modelliamo).
- Siccome **PHP è un linguaggio sincrono**, per implementare flussi asincroni necessita di un sistema di code e/o librerie aggiuntive, aggiungendo, di fatto, ulteriore complessità al nostro sistema.
- **Allineamento culturale:** Il team deve essere disposto ad utilizzare questo pattern.