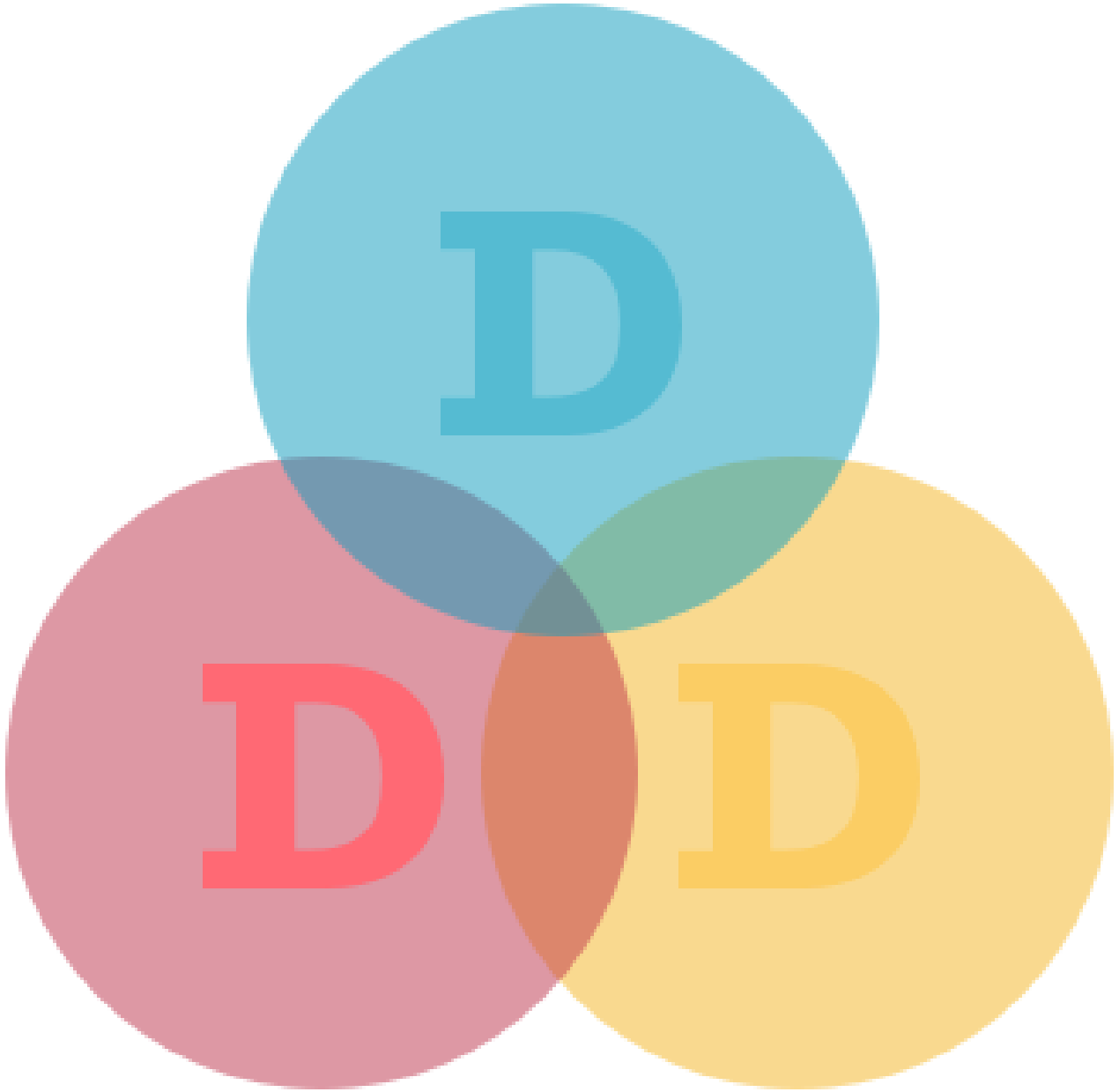

D
D
D

Introduzione a CQRS

CQRS

CQRS (Command and Query Responsibility Segregation) è uno schema architetturale che separa i modelli per la lettura e la scrittura dei dati.

Il termine correlato Command Query Separation (CQS) è stato originariamente definito da Bertrand Meyer nel suo libro Object Oriented Software Construction.

L'idea di base è che è possibile dividere le operazioni di un sistema in due categorie nettamente separate:

- **Query: restituiscono un risultato, non modificano lo stato del sistema e sono prive di effetti collaterali.**

- **Comandi: modificano lo stato di un sistema.**

CQRS

CQRS (Command Query Responsibility Segregation) è un modello semplice che si può applicare a un contesto limitato.

Si tratta di metodi all'interno dello stesso oggetto essendo query o comandi che vengono lanciati.

Ogni metodo restituisce lo stato oppure modifica lo stato, ma non esegue entrambe le operazioni.

Anche un singolo oggetto di schema di repository può essere conforme a CQS.

CQS può essere considerato un principio fondamentale per CQRS.

CQRS

Command and Query Responsibility Segregation (CQRS) è stato introdotto da Greg Young e fortemente promosso da Udi Dahan e altri esperti.

È basato sul principio CQS, sebbene sia più dettagliato.

Può essere considerato uno schema basato sui comandi e sugli eventi, oltre che facoltativamente sui messaggi asincroni.

In molti casi, CQRS è correlato a scenari più avanzati, come l'uso di un database per le operazioni di lettura (query) diverso da quello per le operazioni di scrittura (aggiornamenti).

CQRS

Inoltre, un sistema CQRS più evoluto potrebbe implementare la determinazione dell'origine degli eventi per il database degli aggiornamenti, consentendo di archiviare solo gli eventi nel modello di dominio, anziché archiviare i dati sullo stato corrente.

L'aspetto di separazione di CQRS viene ottenuto mediante il raggruppamento delle operazioni di query in un livello e dei comandi in un altro livello.

Ogni livello dispone di un proprio modello di dati (si noti che parliamo di modello, non necessariamente di un database diverso) e viene creato tramite una specifica combinazione di schemi e tecnologie.

CQRS

Cosa ancora più importante, i due livelli possono essere contenuti nello stesso layer.

In alternativa, possono essere implementati in diversi microservizi o processi, in modo da poter essere ottimizzati e scalati in orizzontale separatamente senza influire l'uno sull'altro.

CQRS significa creare due oggetti per un'operazione di lettura/scrittura, dove in altri contesti ne è presente uno.

CQRS

Separa il modello di dominio in due modelli: un modello di lettura e un modello di scrittura (a volte chiamato modello transazionale).

Il motivo della separazione è consentire a un modello di servire le esigenze di un singolo contesto senza compromessi.

I due contesti in questione riportano lo stato del dominio e svolgono attività aziendali, note anche come lati di lettura e scrittura.

L'utilizzo di un unico modello per contesti delimitati che hanno esigenze di presentazione complesse e logica di dominio ricca spesso fa sì che quel modello diventi eccessivamente complesso e privo di integrità, generando confusione per gli esperti di dominio e un incubo di manutenzione per gli sviluppatori.

CQRS

Applicando il pattern CQRS, un modello viene diviso in due, consentendo a ciascun modello di essere ottimizzato per servire ogni contesto in modo più efficace.

CQRS non è un'architettura di primo livello; è un modello per la gestione della complessità che può essere applicato a contesti limitati che necessitano di supportare un modello di presentazione non allineato alla struttura del modello transazionale.

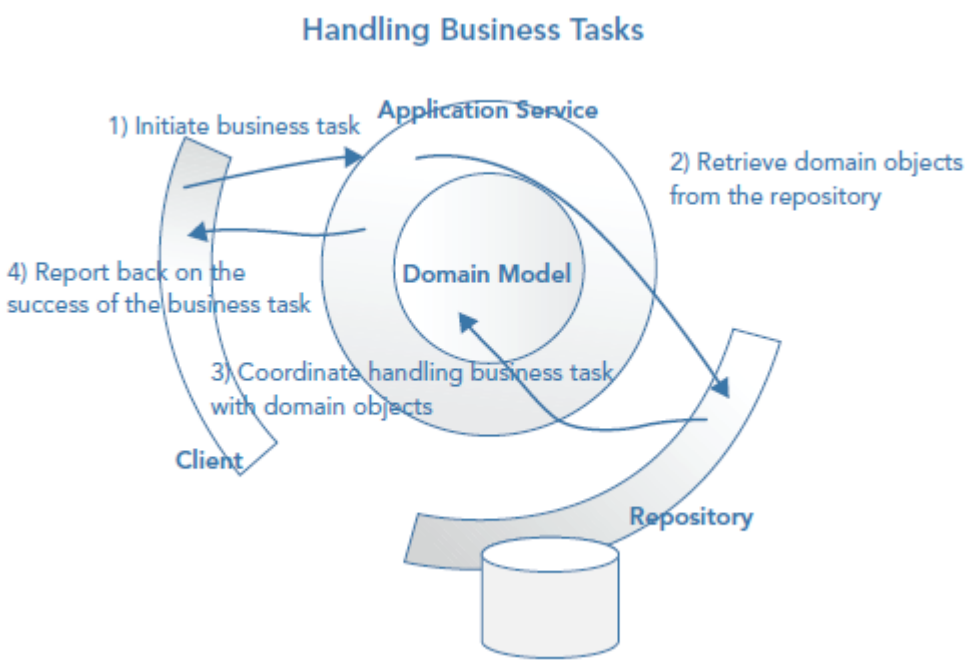
La maggior parte delle applicazioni web vede una disparità tra query e comandi.

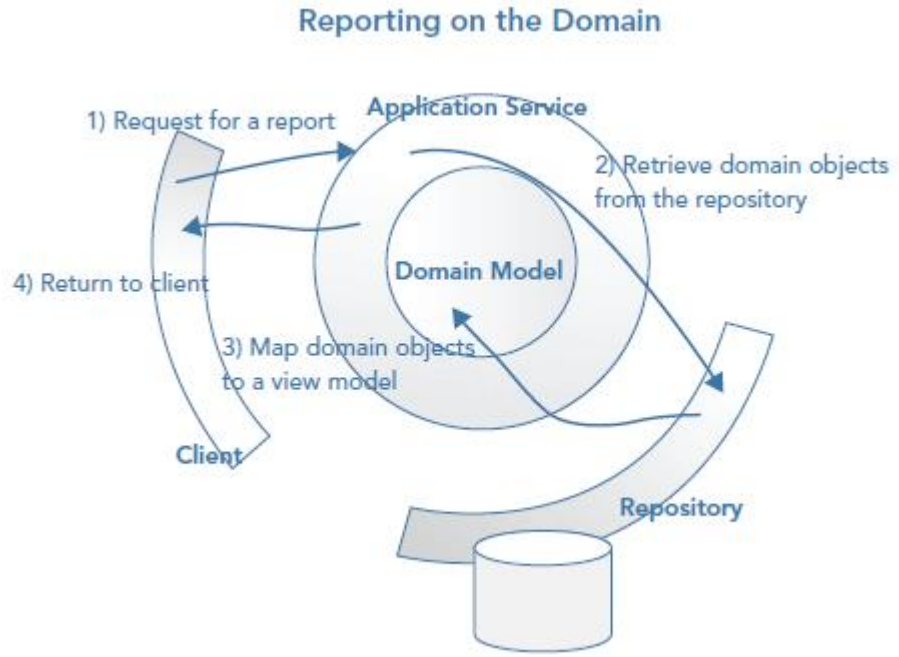
CQRS divide questi due e consente di ottimizzare i lati senza compromessi.

LE SFIDE DI MANTENERE UN UNICO MODELLO PER DUE CONTESTI

LE SFIDE DI MANTENERE UN UNICO MODELLO PER DUE CONTESTI

La figura sottostante mostra una tipica architettura a strati di un contesto delimitato.





Nel cuore di questa architettura si trova un modello di dominio.

Il modello di dominio viene creato per applicare le invarianti del dominio durante la gestione delle operazioni transazionali.

LE SFIDE DI MANTENERE UN UNICO MODELLO PER DUE CONTESTI

Il modello è composto da piccoli raggruppamenti aggregati di oggetti di dominio costruiti per coerenza e che esprimono le regole e la logica del dominio.

Le esigenze di reportistica di un'applicazione, tuttavia, potrebbero non essere allineate con la struttura degli aggregati, con il risultato che i servizi dell'applicazione necessitano di caricare molti aggregati diversi per costruire modelli di visualizzazione che possono contenere solo un sottoinsieme dei dati recuperati all'interno delle istanze aggregate .

Questa generazione di visualizzazioni può diventare rapidamente complessa e difficile da mantenere e negli scenari peggiori può rallentare il sistema.

LE SFIDE DI MANTENERE UN UNICO MODELLO PER DUE CONTESTI

Per supportare la generazione di viste, i modelli di dominio devono esporre lo stato interno e devono essere adornati con proprietà di presentazione che hanno poco a che fare con le invarianti del dominio.

I repository spesso contengono molti metodi extra sul contratto per supportare le esigenze di presentazione come paging, query e ricerca di testo libero.

Poiché il lato di lettura di un'applicazione viene in genere utilizzato più frequentemente del lato di scrittura, gli utenti spesso cercano miglioramenti nella generazione di report.

LE SFIDE DI MANTENERE UN UNICO MODELLO PER DUE CONTESTI

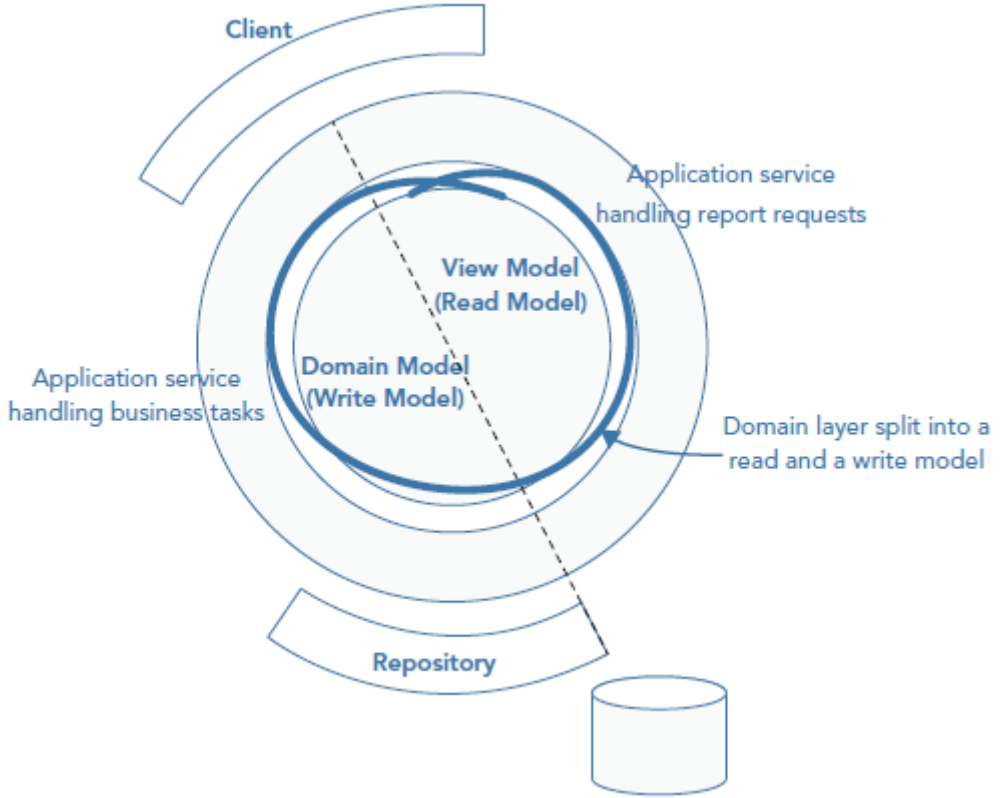
Per cercare di semplificare e migliorare le prestazioni del lato query, il modello viene compromesso.

Gli aggregati vengono uniti e il caricamento lento viene utilizzato per impedire l'estrazione di dati non necessari per le esigenze di elaborazione delle attività aziendali transazionali, ma per scopi di presentazione.

Ciò porta a un unico modello pieno di compromessi ed è inferiore agli standard sia per la lettura che per la scrittura.

UN'ARCHITETTURA MIGLIORE PER CONTESTI DELIMITATI COMPLESSI

La Figura sottostante mostra un'architettura che utilizza il pattern CQRS.



Tratta le esigenze dei due contesti in conflitto, ovvero legge e scrive, separatamente, fornendo due modelli invece di uno.

UN'ARCHITETTURA MIGLIORE PER CONTESTI DELIMITATI COMPLESSI

Ogni modello può ora essere ottimizzato per il contesto specifico che serve pur mantenendo la sua integrità concettuale.

In un certo senso stai applicando il modello di contesto delimitato a un livello inferiore associando un modello per il contesto di lettura e un modello separato per il contesto di scrittura.

La figura mostra la separazione tra i comandi; la responsabilità di adempiere alle attività aziendali, invocate da un cliente, e le domande e la responsabilità di soddisfare i rapporti, che sono richieste da un cliente.

Lo stesso archivio dati viene utilizzato sia per il lato di lettura che per quello di scrittura dell'architettura.

Questo non è obbligatorio; un archivio di lettura separato può essere impiegato per scalare il lato di lettura.

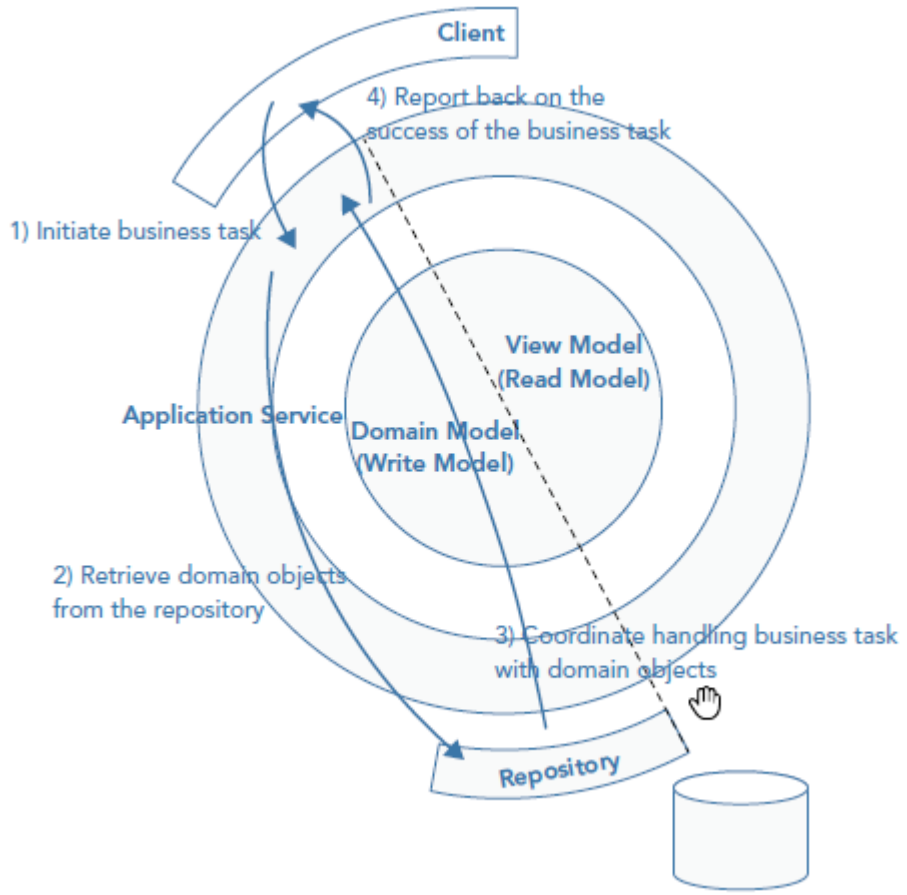
THE COMMAND SIDE: BUSINESS TASKS

UN'ARCHITETTURA MIGLIORE PER CONTESTI DELIMITATI COMPLESSI

Il «command site» dell'architettura si occupa del rispetto delle regole del dominio e rappresenta la logica del dominio che soddisfa le attività aziendali.

L'architettura mostrata nella figura a lato a prima vista è la stessa del tipico approccio a strati.

Tuttavia, il commands side non supporta le query e qualsiasi risposta è solo un riconoscimento del successo dell'attività aziendale avviata da un client.



Explicitly Modeling Intent

Un command è un business task, un caso d'uso di un sistema e risiede all'interno del livello dell'applicazione.

I command andrebbero definiti nella lingua dell'azienda.

Questo non è UL; è il linguaggio che cattura i comportamenti dei sistemi piuttosto che i termini e i concetti del modello di dominio.

In genere, se stai seguendo un approccio BDD, i comandi provengono dai casi d'uso e dalle storie che produci.

Dovresti modellare i comandi come verbi anziché come nomi.

Dovrebbero catturare esplicitamente l'intento dell'utente.

Explicitly Modeling Intent

Un esempio di command è mostrato nel seguente esempio.

```
public class CustomerWantsToRedeemAGiftCertificate
{
public CustomerWantsToRedeemAGiftCertificate(Guid accountId,
string giftCertificate)
{
AccountId = accountId;
GiftCertificate = giftCertificate;
}
public Guid AccountId { get; private set; }
public string GiftCertificate { get; private set; }
}
```

Un command è un semplice oggetto di trasferimento dati (DTO) con una semplice convalida dei parametri.

Il nome del command rivela l'intento di un utente; in questo caso, spetta al cliente riscattare un buono regalo.

Explicitly Modeling Intent

Il command rappresenta una richiesta per l'esecuzione di un'attività aziendale ed è quindi scritto nel tempo presente (ad esempio: voglio fare qualcosa) in contrapposizione agli eventi di dominio, che sono scritti al passato (ad esempio: è successo qualcosa).

Un modello libero dalle distrazioni della presentazione

Un modello che soddisfa sia le esigenze di presentazione che quelle di transizione di un'applicazione spesso assomiglia alle interfacce utente di tale applicazione.

La forma degli aggregati viene trasformata dalla gestione degli invarianti in strutture che corrispondono all'interfaccia utente.

Ad esempio, prendete il mockup di un'interfaccia utente come in figura.

Questa schermata simile a un dashboard presenta vari attributi di un cliente.

Un modello libero dalle distrazioni della presentazione

Il Listato seguente mostra il tipo di oggetto di dominio che viene spesso creato per soddisfare le esigenze della presentazione implementando al contempo la logica del dominio.

```
public class Customer
{
// ...
public ContactDetails ContactDetails { get; private set; }
public LoyaltyStatus LoyaltyStatus { get; private set; }
public Money GiftCertBalance { get; private set; }
public IEnumerable<Address> AddressBook { get; private set; }
}
```

Un modello libero dalle distrazioni della presentazione

È ora possibile generare facilmente una vista per il modello di presentazione perché il modello di dominio è completamente allineato con l'interfaccia utente.

Si tratta semplicemente di recuperare l'aggregato completo del cliente e di mapparlo a un modello di visualizzazione, come dimostrato nel Listato seguente:

```
public class CustomerDashBoardView
{
// ...
public CustomerViewModel Generate(Guid customerId)
{
var customer = _customerRepository.FindBy(customerId);
var customerView = MapCustomerViewModelFrom(customer);
return customerView;
}
}
```

Un modello libero dalle distrazioni della presentazione

Tuttavia, l'aggregato è ora molto grande ed è responsabile di tutto ciò che è associato a un cliente.

L'aggregato è strutturato attorno all'interfaccia utente anziché alle invarianti del dominio.

In altre parole, gli aggregati si basano su di una schermata di un report anziché sul comportamento del dominio.

È possibile evitare questi problemi applicando CQRS e liberando il modello da qualsiasi requisito di presentazione.

Un modello libero dalle distrazioni della presentazione

Nel command model, non vi è alcun vantaggio nella creazione di un unico concetto di cliente, spesso visto come un «code smell» e indicato come una «classe di Dio».

Invece, ci sarà un aggregato responsabile delle regole che governano la lealtà, un aggregato separato per i dettagli del cliente e un terzo per il saldo del buono regalo.

L'UL si adatterà meglio ai comportamenti dell'applicazione rispetto alla schermata dell'interfaccia utente.

Gli esperti di dominio parleranno di comportamento e regole, non dell'interfaccia utente.

Un modello libero dalle distrazioni della presentazione

Senza le sue responsabilità aggiuntive, il command model può essere più piccolo e più focalizzato sul comportamento con i servizi applicativi e le aggregazioni possono diventare più concise.

I repository possono essere notevolmente semplificati poiché il recupero aggregato è limitato dall'ID piuttosto che da una serie di metodi di query come paging e ordinamento.

Gli sviluppatori possono modellare gli aggregati attorno agli invarianti e concentrarsi sul comportamento transazionale senza preoccuparsi della presentazione.

Handling di una Business Request

Un command handler (gestore di comandi) può essere considerato come un «condimento» di un'application service.

Il gestore elabora il comando e contiene la logica per orchestrare il completamento di un'attività.

Questa logica può includere la delega al modello di dominio, la persistenza e il recupero e la chiamata a servizi di infrastruttura come i client di posta elettronica dei gateway di pagamento.

Un gestore di comandi restituisce solo un riconoscimento del successo o fallimento di un comando;

Non dovrebbe essere usato per query o report nello stato del dominio.

Il Listato seguente presenta un esempio di un gestore di comandi.

```
public class CreateOrUpdateCategoryHandler
{
// ...
public ICommandResult Execute(CreateOrUpdateCategoryCommand command)
{
var category = new Category
{
CategoryId = command.CategoryId,
Name = command.Name,
Description = command.Description
};
if (category.CategoryId == 0)
categoryRepository.Add(category);
else
categoryRepository.Update(category);
unitOfWork.Commit();
return new CommandResult(true);
}
}
```

Handling di una Business Request

Poiché un domain model sul command side è progettato per implementare regole e logica di dominio, non è necessario che contenga proprietà di presentazione non necessarie.

Gli aggregati DDD supportano l'elaborazione dei comandi piuttosto che modellare la vita reale.

Gli handlers aiutare a concentrare gli aggregati sul comportamento e sulle invarianti piuttosto che sulla vita reale.

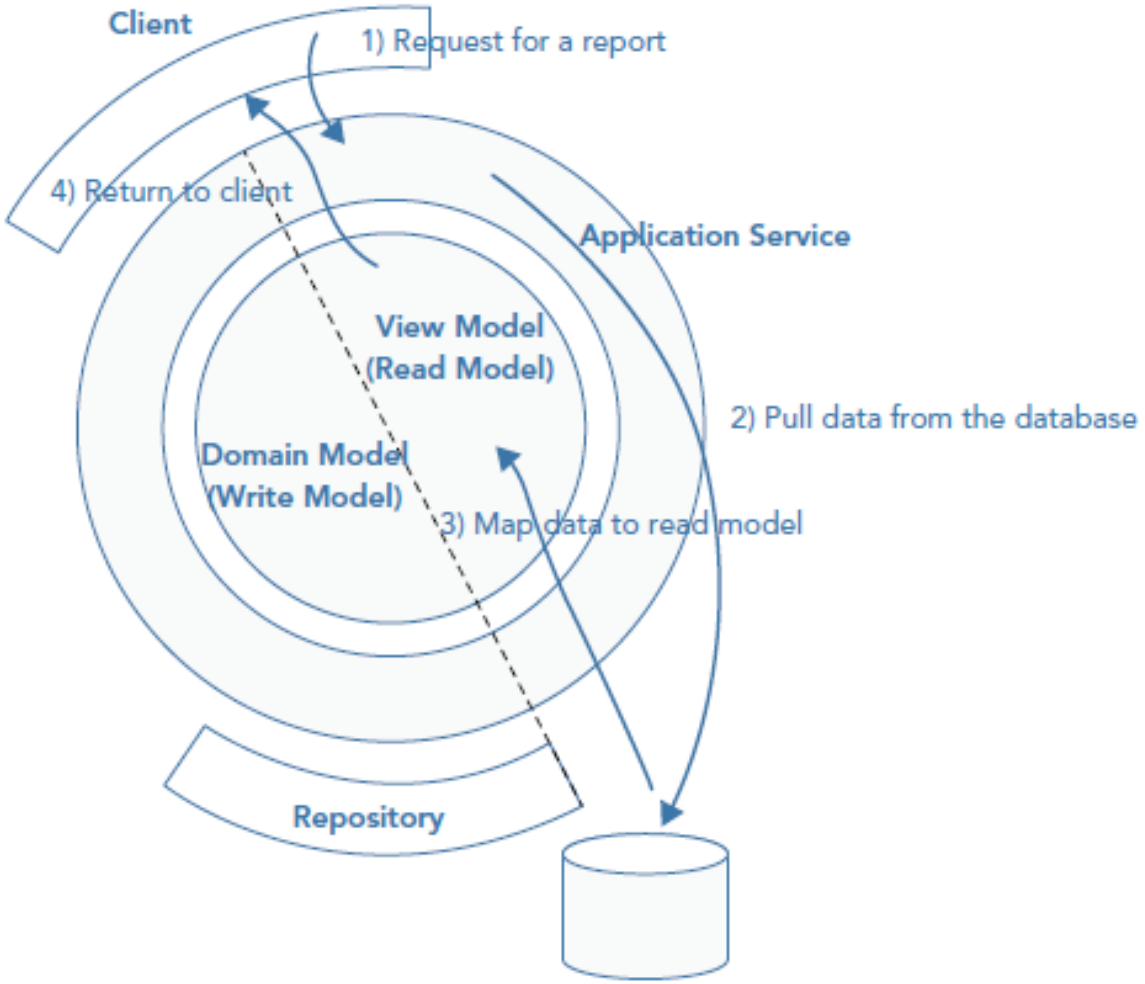
I comandi specifici non richiedono entità di dominio complete quando li gestiamo (ad esempio: non hanno bisogno del nome del cliente quando esegui un'azione sull'aggregazione dei clienti).

L'entità cliente non ha senso e ciò aiuta a mantenere gli aggregati piccoli e a modellare aggregati più piccoli con meno associazioni.

THE QUERY SIDE: DOMAIN REPORTING

THE QUERY SIDE: DOMAIN REPORTING

L'architettura del query side, come mostrato nella figura sottostante ha a che fare con il reporting sul dominio.



THE QUERY SIDE: DOMAIN REPORTING

Gli oggetti restituiti dal query side sono semplici DTO view models adattati alle esigenze specifiche della view.

Il domain model per il command side non è richiesto, poiché è possibile generare una view direttamente dall'archivio dati.

Il query side non ha bisogno di creare un'astrazione sull'archivio di persistenza, quindi un repository in questo contesto non ha senso.

Un framework di persistenza (o librerie leggere) come ADO.NET dovrebbero essere usati qui.

THE QUERY SIDE: DOMAIN REPORTING

Poiché il modello di lettura è ancora all'interno del domain layer, è in grado di utilizzare oggetti di dominio dal read side per eseguire calcoli se questi dati non sono precalcolati all'interno dell'archivio dati.

In genere, le classi specification sono state impiegate per fornire una risposta alle proprietà del view model sulla base di alcuni dati estratti dal database.

Report mappati direttamente al modello di dati

Il read side dell'architettura mappa le richieste di report modellate come view models direttamente al data model, bypassando completamente il command model.

Le views possono essere create all'interno del modello di dati per ogni schermata o report dell'interfaccia utente.

Ciò si traduce in viste sintetiche, che sono veloci da recuperare, semplici quando si mappano i dati grezzi per i view models e in grado di gestire l'impaginazione e l'ordinamento e le ricerche di testo libero.

Report mappati direttamente al modello di dati

Se il command side non precalcola un valore richiesto, utilizzare una specifica o un o un servizio di dominio per calcolare il valore al volo come mostrato nel listato seguente.

```
public class OrderQuery
{
// ...
public OrderViewModel Generate(Guid customerId)
{
var customer = _customerRepository.FindBy(customerId);
var customerView = MapCustomerViewModelFrom(customer);
customerView.IsInfluential =
_influentialSpec.SatisfiedBy(customer);
return customerView;
}
}
```

Report mappati direttamente al modello di dati

È possibile utilizzare un micro Object Relational Mapper (ORM) nel servizio di query privo di qualsiasi logica, tranne quella di estrarre dati e mappare a DTO e delegare a specifiche o servizi di dominio per decisioni basate sullo stato salvato.

Con un micro ORM si intende qualcosa di leggero in grado di mappare rapidamente i dati grezzi su di un DTO view model in modo tale che si semplice il read side.

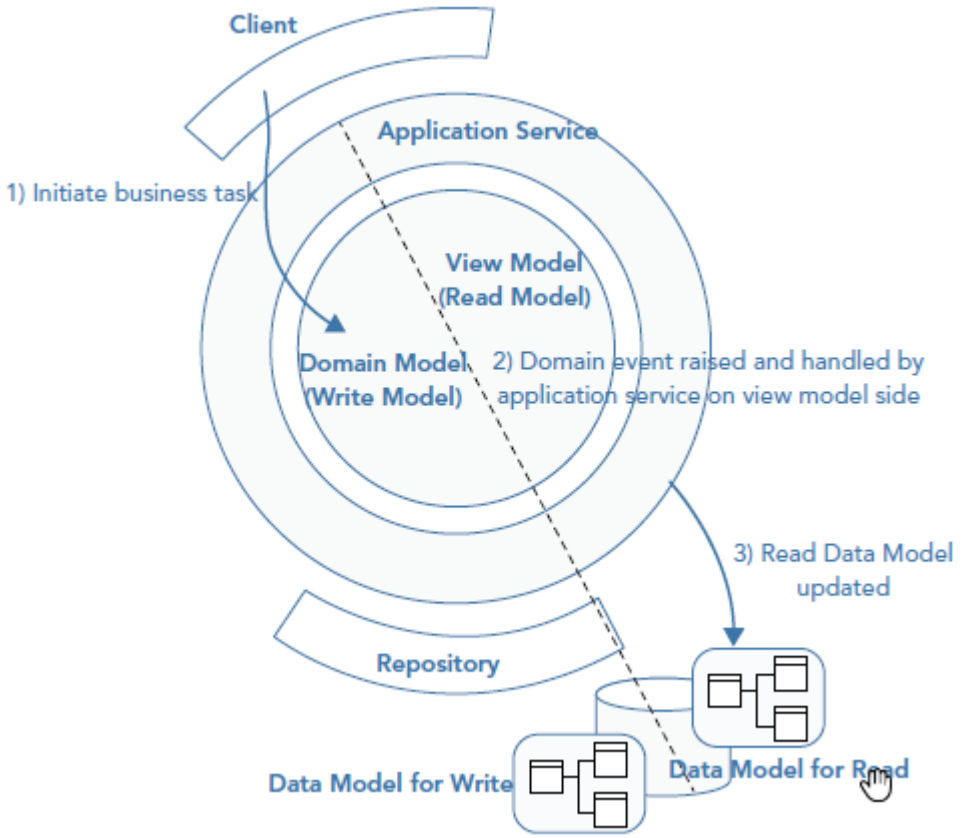
Viste materializzate costruite da eventi di dominio

Puoi andare oltre con la segregazione di lettura e scrittura utilizzando uno schema di dati diverso.

È possibile creare un modello di lettura dagli eventi di dominio generati dai commands.

E' quindi possibile utilizzare questi eventi per creare viste materializzate, come mostrato in figura.

Un read model può essere denormalizzato e ottimizzato per l'interrogazione, inclusi i dati precalcolati.



Report mappati direttamente al modello di dati

Il Listato seguente mostra come un'azione sul command model porta a un evento di dominio che viene generato e quindi persistito aggiornando il read model.

Ciò può accadere all'interno della stessa transazione e dello stesso database.

L'aggiornamento del modello di lettura fuori processo può consentire alla tua applicazione di scalare.

```
public class ModifiyCategoryHander
{
public void Execute(ModifyCategoryCommand command)
{
var category = _catalogueRepository.FindBy(commmand.Id);
using (DomainEvents.Register<CategoryUpdated>(onCategoryUpdated))
{
category.Update(command);
}
}
private void onCategoryUpdated(CategoryUpdated @event)
{
_catalogueViewModel.Update(CategoryUpdated);
}
}
```


I PREGIUDIZI DI CQRS

I PREGIUDIZI DI CQRS

Ci sono molte idee sbagliate sul modello di CQRS.

E' semplice ma va definito se è il caso di utilizzarlo come un modello specifico per un contesto specifico.

Se si ha un'infarinatura di base su CQR, si potrebbe pensare di dover utilizzare framework di messaggistica pesanti o che sia necessario uno sforzo di programmazione maggiore per rendere coerente il read store.

Questa convinzione è errata in quanto possiamo utilizzare queste tecniche per estendere il pattern CQRS per scalare l'applicazione quando necessario.

Di seguito elencheremo molti dei pregiudizi errati e più comuni sul modello CQRS.

CQRS è difficile ?

CQRS nella sua applicazione base è un modello semplice.

A livello base è un'implementazione dell'SRP (Single Responsibility Principle) applicato a livello del domain model layer.

È utile per risolvere la complessità che sorge quando un modello di presentazione non è allineato con un modello transazionale.

CQRS non necessita di framework, database multipli o design patterns.

Afferma solo che i due contesti (query e command) dovrebbero essere gestiti separatamente al fine di renderli più efficaci.

È un cambiamento concettuale e mentale piuttosto che una raccolta di schemi e principi complessi da adottare.

CQRS è alla fine consistente ?

La consistenza è la pratica di avere un read model aggiornato «out of process» e «in modo asincrono» rispetto a quanto accade nel modello transazionale.

Questo non è un prerequisito di CQRS, ma viene spesso utilizzato per consentire la scalabilità del read side di un modello.

CQRS non richiede la consistenza.

È possibile utilizzare lo stesso database e la stessa transazione per aggiornare lo schema del modello di lettura.

In effetti, l'archivio di lettura della tua applicazione potrebbe essere già alla fine coerente se si utilizza utilizzi un sistema di caching.

CQRS è alla fine coerente ?

Se si adotta il modello CQRS e, a causa di complessità che provocano problemi di disallineamento tra presentazione e transazione, si hanno problemi di performances bisognerebbe accettare il compromesso di non essere «essere immediatamente coerente» e passare a «eventualmente coerente».

Infatti potrebbero esservi sovraccarichi (in architetture particolarmente complesse) nel cercare di utilizzare archivi di lettura coerenti.

I modelli devono avere come base gli eventi ?

L'utilizzo dell'«event sourcing» è un metodo efficace per costruire sia i modelli di lettura che quelli di scrittura; tuttavia, non vi è alcun prerequisito per l'utilizzo del «event sourcing» o di «fact domain events» con CQRS.

L'«event sourcing» è una soluzione per garantire che il tuo audit trail (meccanismo di controllo di un sistema che permette, per i dati imputati e successivamente processati dal sistema, la tracciabilità a ritroso al dato originale.) sia accurato.

Inoltre semplifica la creazione del modello di lettura perché consente di creare qualsiasi proiezione si voglia dai dati storici degli eventi.

I commands dovrebbero essere asincroni ?

CQRS non pretende che i commands vengano inviati in modalità «fire-and-forget».

Per i domini altamente collaborativi, in cui più utenti stanno apportando modifiche agli stessi dati, i comandi asincroni hanno senso.

Ciò consente loro di essere gestiti a turno e consente all'applicazione di scalare e e bilanciare il carico.

Tuttavia, i comandi che non restituiscono un esito positivo o negativo richiedono altri modi per aggiornare l'utente sul successo di un'azione.

Ciò potrebbe avvenire tramite e-mail o un comportamento aggiuntivo che gestisce i messaggi falliti.

CQRS funziona solo con i sistemi di messaggistica ?

Se si cerca di applicare in modo asincrono comandi di lettura o elaborazione coerenti, è probabilmente una buona idea utilizzare un framework di messaggistica.

Tuttavia aggiungere un sistema di messaggistica ad un'applicazione può essere solo una inutile complicanza.

È necessario utilizzare gli eventi di dominio con CQRS ?

L'utilizzo di eventi per costruire un modello di lettura materializzato è un metodo efficace per mantenere separati i modelli di lettura e scrittura; tuttavia, non è fondamentale ed è possibile utilizzare altri .

Gli aggregati possono rivelare uno stato utilizzando il pattern memento.

È inoltre possibile utilizzare alcuni modelli non a eventi per fornire informazioni sulla presentazione direttamente dagli oggetti del dominio senza che vi siano particolari problemi.

Infine, è possibile creare viste basate sul modello di dati relazionali del modello di scrittura.