CQRS ed Event Sourcing

Workshop pratico per sviluppatori PHP

Cos’è CQRS

- CQRS (Command/Query Responsability Segregation) è un pattern utilizzato per la modellazione del dominio di un software

Cos’è CQRS

- 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

- **Comandi e query dovrebbero essere tenuti separati**. Nel nostro dominio ogni modello dovrebbe essere pensato per la lettura o per la scrittura, in scenari complessi infatti, un modello che si occupa sia di scrittura che di lettura potrebbe facilmente crescere a dismisura diventando presto difficile da gestire e quindi costoso

Cos’è Event Sourcing

- è un pattern che **utilizza gli eventi come strumento per modellare la logica di business**.
  Lo stato dell’applicazione e i suoi cambiamenti vengono rappresentati come una sequenza di eventi salvati nella base di dati, chiamata event store

Cos’è Event Sourcing

- Non esistono operazioni di modifica o cancellazione di eventi, siamo in append mode e tutto quello che è successo nel nostro dominio è rappresentato da questa sequenza

Cos’è Event Sourcing

- Cosa succede nella realtà quando commettiamo un errore ?

Cos’è Event Sourcing

- Differentemente da un comando che rappresenta un azione che si sta compiendo nel presente, come ad esempio “approve a loan”, **l’evento rappresenta un’azione passata ed è descritta come tale, come ad esempio “loan approved”**.

Comando

- Il comando rappresenta un’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

- Il comando è dunque il messaggio inviato all’aggregato radice (Aggregate Root – AR) e contiene i dati necessari ad eseguire la logica di business rappresentata dall’azione.

Evento

- L’evento rappresenta una cosa successa nel nostro sistema: in OOP può essere rappresentato come un oggetto immutabile il cui nome descrive l’azione successa e il suo stato rappresenta il nuovo stato persistito

Comando + Evento

- Ad ogni comando inviato all’aggregato radice, viene emesso e salvato un evento che rappresenta il cambiamento di stato dovuto all’azione richiesta

Aggregato

- Un aggregato rappresenta il cuore del nostro dominio o di una parte di esso. Può essere visto come un grafo di oggetti (o anche un unico oggetto) che definisce logiche di business

- Generalmente si individua un singolo oggetto del grafo, detto aggregato root, che ne espone tutto il comportamento. L’aggregato root è l’unico punto d’accesso all’intero aggregato

Aggregato

- Questo oggetto ha la responsabilità di ricevere il comando, eseguire logica di business rispettando le invarianti e infine generare eventi che verrano poi salvati sull'event store

- Le invarianti rappresentano dei vincoli di dominio che impedisco all'aggregato di assumere degli stati inconsistenti ovvero stati che non esistono nel dominio

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

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 lettetura

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

Quando ?

- Domini complessi ovvero con logiche complesse, flussi di business complessi e quindi domini che richiedono una buona risposta al cambiamento

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 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

- business intelligence
- **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 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

- **Broadway**: Con il componente per il testing che mette a disposizione, possiamo testare unitariamente e molto facilmente tutta la logica di dominio

- 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

Svantaggi

- Curva apprendimento lunga
- E’ un pattern non molto comune ed ancora non molto utilizzato dalla comunità PHP
- Diventa tremendamente 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 (occhio 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