



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA - SCIENZA e INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA  
INFORMATICA

Analisi, Progettazione e Distribuzione in  
Cloud di applicativo multiplatforma  
per l'organizzazione di eventi condivisi e  
la condivisione multimediale automatica  
in tempo reale

Relatore:  
Chiar.mo Prof.  
Michele Colajanni

Presentata da:  
Giacomo Romanini

---

Sessione Luglio 2025

Anno Accademico 2025/2026

# Abstract

Lo sviluppo di un applicativo multiplatforma diretto all'organizzazione di eventi condivisi, caratterizzato in particolare dalla condivisione multimediale in tempo reale, richiede opportune capacità di scalabilità, atte a garantire una risposta efficace anche con alti volumi di richieste, offrendo prestazioni ottimali. Le tecnologie cloud, con la loro disponibilità pressoché illimitata di risorse e la completa e continua garanzia di manutenzione, offrono l'architettura ideale per il supporto di simili progetti, anche con fondi limitati.

Tuttavia, l'integrazione tra la logica applicativa ed i molteplici servizi cloud, insieme alla gestione delle loro interazioni reciproche, comporta sfide specifiche, in particolare legate all'ottimizzazione di tutte le risorse. L'individuazione e la selezione delle soluzioni tecnologiche più adatte per ogni obiettivo, così come l'adozione delle migliori pratiche progettuali, devono procedere parallelamente con lo sviluppo del codice, al fine di sfruttare efficacemente le potenzialità offerte.

In tale prospettiva, questa tesi illustra le scelte progettuali ed implementative adottate nello sviluppo dell'applicativo in questione, evidenziando l'impatto dell'integrazione delle risorse cloud sul risultato finale.

# Indice

<b>Introduzione</b>	<b>1</b>
Organizzazione dei capitoli . . . . .	3
<b>1 La gestione della persistenza</b>	<b>4</b>
1.1 L'analisi per l'identificazione del database . . . . .	6
1.1.1 Le proprietà dei database relazionali . . . . .	7
1.1.2 Le proprietà dei database non relazionali . . . . .	8
1.1.3 L'impatto delle relazioni delle entità sulle prestazioni . . . . .	9
1.1.4 Analisi del dominio . . . . .	11
1.2 L'implementazione del database . . . . .	14
1.2.1 La scelta del database . . . . .	15
1.2.2 La definizione delle classi . . . . .	20
1.2.3 L'integrazione con le Azure Functions: il framework .Net e l'eventual consistency . . . . .	21
1.2.4 L' integrazione con C# . . . . .	22

# Introduzione

In un contesto sociale sempre più connesso, la crescente quantità di contatti, la rapidità delle comunicazioni e l'accesso universale alle informazioni rendono la ricerca, l'organizzazione e la partecipazione ad eventi estremamente facile, ma al contempo generano un ambiente frenetico e spesso dispersivo.

Risulta infatti difficile seguire tutte le opportunità a cui si potrebbe partecipare, considerando le numerose occasioni che si presentano quotidianamente. Basti pensare, ad esempio, alle riunioni di lavoro, alle serate con amici, agli appuntamenti informali per un caffè, ma anche a eventi più strutturati come fiere, convention aziendali, concerti, partite sportive o mostre di artisti che visitano occasionalmente la città.

Questi eventi possono sovrapporsi, causando dimenticanze o conflitti di pianificazione, con il rischio di delusione o frustrazione. Quando si è invitati a un evento, può capitare di essere già impegnati, o di trovarsi in attesa di una conferma da parte di altri contatti. In questi casi, la gestione degli impegni diventa complessa: spesso si conferma la partecipazione senza considerare possibili sovrapposizioni, o dimenticandosi, per poi dover scegliere e disdire all'ultimo momento.

D'altra parte, anche quando si desidera proporre un evento, la ricerca di un'attività interessante può diventare un compito arduo, con la necessità di consultare numerosi profili social di locali e attività, senza avere inoltre la certezza che gli altri siano disponibili. Tali problemi si acuiscono ulteriormente quando si tratta di organizzare eventi di gruppo, dove bisogna allineare gli impegni di più persone.

In questo contesto, emergono la necessità e l'opportunità di sviluppare uno strumento che semplifichi la proposta e la gestione degli eventi, separando il momento della proposta da quello della conferma di partecipazione. In tal modo, gli utenti possono valutare la disponibilità degli altri prima di impegnarsi definitivamente, facilitando in contemporanea sia l'invito sia la partecipazione.

In risposta a tali richieste è stata creata Wyd, un'applicazione che permette agli utenti di organizzare i propri impegni, siano essi confermati oppure proposti. Essa permette anche di rendere più intuitiva la ricerca di eventi attraverso la creazione di uno spazio virtuale centralizzato dove gli utenti possano pubblicare e consultare tutti gli eventi disponibili, diminuendo l'eventualità di perderne qualcuno. La funzionalità chiave di questo progetto si fonda sull'idea di affiancare alla tradizionale agenda degli impegni confermati un calendario separato, che mostri tutti gli eventi a cui si potrebbe partecipare.

Una volta confermata la partecipazione a un evento, questo verrà spostato automaticamente nell'agenda personale dell'utente. Gli eventi creati potranno essere condivisi con persone o gruppi, permettendo di visualizzare le conferme di partecipazione. Considerando l'importanza della condivisione di contenuti multimediali, questo progetto prevede la possibilità di condividere foto e video con tutti i partecipanti all'evento, attraverso la generazione di link per applicazioni esterne o grazie all'ausilio di gruppi di profili. Al termine dell'evento, l'applicazione carica automaticamente le foto scattate durante l'evento, per allegarle a seguito della conferma dell'utente.



Figura 1: Il logo di Wyd

La realizzazione di un progetto come Wyd implica la risoluzione e la gestione di diverse problematiche tecniche. In primo luogo, la stabilità del programma deve essere garantita da un'infrastruttura affidabile e scalabile. La persistenza deve essere modellata per fornire alte prestazioni sia in lettura che in scrittura indipendentemente dalla quantità delle richieste, rimanendo però aggiornata e coerente. La funzionalità di condivisione degli eventi richiede inoltre l'aggiornamento in tempo reale verso tutti gli utenti coinvolti. Infine, il caricamento ed il salvataggio delle foto aggiungono la necessità di gestire richieste di archiviazione di dimensioni significative.

## Organizzazione dei capitoli

Il seguente elaborato è suddiviso in cinque capitoli.

Nel primo capitolo si affronta la fase di analisi delle funzionalità, durante la quale, partendo dall'idea astratta iniziale, si definiscono i requisiti e le necessità del sistema, per poi creare la struttura generale ad alto livello dell'applicazione.

Nel secondo capitolo si affrontano le principali scelte architettureali e di sviluppo che hanno portato a definire la struttura centrale dell'applicazione.

Il terzo capitolo osserva lo studio effettuato per gestire la memoria, in quanto fattore che più incide sulle prestazioni. Particolare attenzione è stata dedicata, infatti, a determinare le tecnologie e i metodi che meglio corrispondono alle esigenze derivate dal salvataggio e dall'interazione logica degli elementi.

Il quarto capitolo si concentra sulle scelte implementative adottate per l'inserimento le funzionalità legate alla gestione delle immagini, che, oltre ad introdurre problematiche impattanti sia sulle dimensioni delle richieste sia sull'integrazione con la persistenza, richiedono l'automatizzazione del recupero delle immagini.

Infine, nel quinto capitolo, verranno analizzati e discussi i risultati ottenuti testando il sistema.

## Capitolo 1

# La gestione della persistenza

Nel contesto di un sistema che prevede la gestione di eventi e impegni, è essenziale garantire all'utente di poter accedere in qualsiasi momento alla propria agenda, visualizzando gli appuntamenti più urgenti e pianificando efficacemente il proprio tempo. Deve perciò essere possibile trovare e mostrare nel minor tempo possibile i dati relativi all'agenda. Per soddisfare questo requisito, il recupero e la visualizzazione dei dati devono avvenire con la massima rapidità possibile, riducendo i tempi di latenza e ottimizzando il flusso di interazione con l'interfaccia utente.

L'adozione di un meccanismo di salvataggio locale sui dispositivi offre il vantaggio di migliorare le prestazioni, consentendo un accesso immediato alle informazioni senza dover effettuare continue richieste al server remoto. Tuttavia questa soluzione rimane parziale, in quanto non garantisce una persistenza a lungo termine dei dati, né assicura la loro disponibilità in ogni momento, per essere in grado di sincronizzare più dispositivi. Per superare tali criticità, è necessario definire una strategia di gestione della memoria che preveda una fonte di dati centrale e autorevole, alla quale tutti i dispositivi possano fare riferimento per recuperare e aggiornare le informazioni in modo coerente e affidabile.

Un sistema di persistenza efficace deve quindi prevedere un meccanismo di sincronizzazione tra i dati salvati localmente e la loro controparte ufficiale memorizzata nel database principale. Questo processo deve essere progettato in modo da garantire integrità, coerenza e scalabilità nelle interazioni, per essere resistente anche in presenza di un volume significativo di richieste concorrenti. La struttura del sistema di memorizzazione deve inoltre essere progettata tenendo conto del dominio applicativo e delle esigenze specifiche

di utilizzo, al fine di assicurare un bilanciamento ottimale tra efficienza e robustezza operativa.

Oltre alla gestione della persistenza dei dati per il singolo utente, è necessario affrontare il problema della modifica di eventi condivisi. Poiché l'applicazione consente a più utenti di interagire sugli stessi eventi, le modifiche effettuate da un partecipante devono essere propagate in tempo reale agli altri dispositivi coinvolti. Questo introduce la necessità di implementare un sistema di aggiornamento distribuito, in grado di mantenere sincronizzati non solo i dispositivi di un singolo utente, ma anche quelli di tutti gli utenti interessati dalle modifiche.

La gestione dell'accesso ai dati richiede quindi l'implementazione di un'architettura che preveda un punto di riferimento centrale chiaro e affidabile, capace di fungere da fonte primaria delle informazioni. Al tempo stesso, l'utilizzo di copie locali dei dati sui dispositivi client consente di ridurre l'impatto delle latenze di rete, migliorando la reattività dell'interfaccia e offrendo un'esperienza utente più fluida. Tuttavia, questa scelta introduce la necessità di gestire due livelli distinti di responsabilità: da un lato, il client deve occuparsi di mantenere aggiornati i dati memorizzati localmente, mentre il server deve garantire la corretta distribuzione delle modifiche agli altri dispositivi interessati. La sincronizzazione e la gestione delle versioni dei dati diventano quindi elementi chiave per assicurare la coerenza del sistema e prevenire eventuali conflitti tra modifiche concorrenti.



## 1.1 L'analisi per l'identificazione del database

Nell'implementazione di applicazioni scalabili, la gestione del salvataggio dei dati può essere strutturata secondo un modello centralizzato o distribuito, a seconda delle esigenze di affidabilità, scalabilità e prestazioni del sistema. L'adozione di un'architettura di memoria distribuita offre molteplici vantaggi, tra cui una maggiore resilienza ai guasti di una singola fonte, la riduzione del carico di memoria su un'unica risorsa di archiviazione e una migliore scalabilità complessiva del sistema. Tuttavia, questa soluzione introduce una maggiore complessità infrastrutturale, poiché richiede meccanismi avanzati per garantire il recupero, l'affidabilità e la consistenza delle informazioni.

Salvo specifici requisiti che rendano indispensabile la distribuzione totale o parziale della memoria, una strategia basata su un database centralizzato risulta più efficiente dal punto di vista prestazionale e semplifica la gestione complessiva del sistema. L'adozione di un'architettura centralizzata consente infatti di ottimizzare i tempi di accesso ai dati e ridurre la latenza delle operazioni, grazie a una minore complessità di sincronizzazione e di mantenimento della consistenza delle informazioni.

I database non distribuiti si suddividono in due macro categorie principali: relazionali e non relazionali.

I database relazionali si caratterizzano per strutture dati rigide e schematizzate, che consentono di stabilire connessioni tra le diverse entità in tempi estremamente rapidi, garantendo allo stesso tempo operazioni atomiche. Viceversa, i database non relazionali offrono una maggiore flessibilità strutturale, permettendo l'archiviazione di dati eterogenei e adottando un accoppiamento più debole tra i vari oggetti.

La scelta della tipologia di database più appropriata dipende direttamente dalle esigenze specifiche del progetto. Ogni prodotto è stato infatti realizzato per rispondere a una funzionalità specifica, introducendo vantaggi per un determinato caso d'uso ma comportando anche punti deboli, sia in termini di prestazioni che in termini di scalabilità. Determinare il database più adatto alle esigenze dipende quindi non solo dalle proprietà intrinseche della tecnologia, ma soprattutto di come queste riescano a risolvere i particolari problemi che il progetto presenta.

### 1.1.1 Le proprietà dei database relazionali

I database relazionali gestiscono i dati tramite strutture chiamate schemi. Lo schema è una struttura rigida i cui campi e le relative proprietà vengono definite sin dal momento della creazione. Mantengono però un grande potere espressivo, in quanto permettono di descrivere direttamente le relazioni (e le loro proprietà) tra gli oggetti, attraverso la creazione di uno schema dedicato. Ogni elemento ha un identificativo univoco (Primary Key o PK) attraverso cui viene individuato all'interno del suo schema. Se lo schema descrive una relazione, viene identificato tramite una combinazione di identificativi derivati (Foreign Key o FK). Gli aggiornamenti agli schemi avvengono tramite un rigoroso sistema di transazioni. La transazione è il processo attraverso il quale una modifica viene portata a termine, a cui viene riservata per il tempo necessario la risorsa interessata.

Grazie alla struttura statica dei dati, unitamente alle relazioni predefinite del dominio, il tempo di recupero e analisi dei dati risulta altamente ottimizzato. L'utilizzo delle primary e foreign key consente al database di creare automaticamente indici dedicati che consentono l'individuazione di un oggetto in tempi minimi, e facilitano l'incrocio delle informazioni contenute all'interno delle relazioni. L'utilizzo delle transazioni rende i database relazionali in grado di garantire le proprietà di Atomicità, Consistenza, Isolamento e Durabilità (ACID), assicurando un'elevata affidabilità dei dati.

La rigidità del modello non permette però di avere un elemento di uno schema che presenti proprietà diverse da tutti gli altri. Non è quindi supportata l'aggiunta o la modifica di campi all'interno di un oggetto, a meno di non cambiare la definizione dell'intero schema. L'utilizzo delle transazioni introduce inoltre la necessità, durante le operazioni di scrittura, di bloccare temporaneamente le risorse interessate, facendo fallire o aspettare altre richieste simultanee sullo stesso elemento, potenzialmente influenzando le prestazioni complessive del sistema.

Per poter garantire il successo di una transazione il database deve controllare tutte le sessioni con il server, il che comporta una limitazione al numero massimo di connessioni (e quindi richieste) contemporanee possibili. Inoltre gli indici dipendono da tutti gli elementi del database, e allo stesso modo gli schemi delle relazioni sono fortemente accoppiati

con gli schemi delle entità coinvolte. Risulta quindi arduo dividere degli elementi su più tabelle(sharding), operazione necessaria per la distribuzione del database su più server, aumentando di conseguenza la complessità richiesta per essere in grado di scalare orizzontalmente.

### 1.1.2 Le proprietà dei database non relazionali

I database non relazionali, noti anche come NoSQL, si distinguono per l'adozione di modelli di archiviazione dei dati flessibili, che si discostano dalla rigida struttura tabellare dei database relazionali. Questi modelli includono approcci chiave-valore, documento, colonnare e a grafo, ciascuno ottimizzato per specifiche esigenze applicative, come la gestione di file, dati semi-strutturati, o la rappresentazione di relazioni complesse.

La loro caratteristica fondamentale è la capacità di salvare le informazioni in forme variabili, permettendo di modificare la struttura dei dati senza la necessità di riconfigurare lo schema del database. La vera forza dei database NoSQL, tuttavia, risiede nella loro predisposizione alla scalabilità orizzontale. I NoSQL sono infatti progettati per distribuire il carico su più server, consentendo di gestire volumi crescenti di dati e richieste semplicemente aggiungendo nuovi nodi al sistema.

Nonostante i vantaggi in termini di scalabilità, i database non relazionali presentano però alcune limitazioni significative. La più rilevante è la rinuncia alle proprietà ACID, alla quale si contrappone, per favorire la disponibilità e la tolleranza ai partizionamenti, una consistenza finale (eventual consistency), dove le modifiche ai dati si propagano attraverso il sistema in un certo lasso di tempo, piuttosto che essere immediatamente consistenti su tutti i nodi. Questo può portare a letture di dati "stale" (non aggiornati) in determinate circostanze, il che è può essere un problema per applicazioni che richiedono forte consistenza.

### 1.1.3 L'impatto delle relazioni delle entità sulle prestazioni

L'aspetto determinante alla base della scelta del database riguarda la gestione delle relazioni tra le entità. È fondamentale analizzare la distribuzione delle richieste per ciascun elemento e il carico computazionale che ogni operazione comporta. Tra le operazioni più costose in termini di prestazioni, che più ostacola e rallenta il recupero dei dati, vi è l'operazione di unione(join). Durante l'operazione di unione vengono incrociati i dati di vari elementi per restituire un oggetto coerente che presenti tutte le proprietà necessarie, originariamente distribuite in molteplici tabelle.

Nonostante l'esecuzione di join su database sia altamente ottimizzata (particolarmente in quelli relazionali) e facilitata dall'utilizzo di indici, introduce comunque carichi computazionali di grande entità che impattano significativamente sui tempi di risposta delle richieste, crescendo proporzionalmente con la quantità dei dati presenti nel database. Per questo motivo è bene modellare il dominio nell'ottica di ridurre il più possibile le richieste che comportano l'incrocio di dati da tabelle diverse.

Le relazioni tra elementi possono essere classificate in tre categorie principali: di tipo uno a uno, uno a molti, e molti a molti.

Nelle relazioni uno a uno il recupero dei dati è diretto e richiede uno sforzo computazionale limitato. Nei casi uno a molti e molti a molti il reperimento delle informazioni richiede spesso un'operazione di join, ed è quindi bene eseguire un'attenta valutazione delle tipologie di accesso per ottimizzarne le prestazioni. Un'operazione di join è accettabile se il numero delle entità coinvolte è limitato o facilmente reperibile. Altrimenti, una delle strategie possibili per migliorare l'efficienza delle richieste offerte dai database non relazionali è la denormalizzazione delle entità.

La denormalizzazione consiste nel duplicare o incorporare dati correlati all'interno della stessa entità o documento, eliminando la necessità di operazioni di join complesse e costose in fase di lettura. Questo significa che, anziché avere tabelle separate tra due elementi e collegarle tramite chiavi esterne, un database denormalizzato potrebbe memorizzare direttamente un array dei dati del primo all'interno del documento del secondo. Recuperare tutti i dati necessari per una determinata operazione richiede spesso una singola lettura

da un'unica entità, e raramente un'operazione di join, riducendo drasticamente il numero di accessi al disco e le elaborazioni computazionali. Questo è particolarmente vantaggioso in scenari dove le operazioni di lettura sono molto più frequenti di quelle di scrittura.

Per ogni dato duplicato, infatti, si introduce la complessità di dover garantire la coerenza di tali dati attraverso il sistema. Se un'informazione duplicata viene modificata in una delle sue occorrenze, è essenziale che tale modifica si propaghi correttamente a tutte le altre copie per evitare che il database contenga dati incoerenti. Non esiste un meccanismo automatico che ne assicura la coerenza, e la sua creazione può diventare onerosa e complessa, soprattutto in sistemi distribuiti e con elevato volume di scritture.

In una relazione uno a molti, nel caso in cui la richiesta di quell'elemento non sia frequente ma sia invece importante restituire spesso gli elementi a lui collegati, conviene copiare gli oggetti relativi all'interno dell'elemento singolo. L'impostazione inversa, in cui si copia il singolo all'interno dei molti elementi, comporterebbe l'ispezione di tutti i componenti esistenti alla ricerca di quelli che contengono l'elemento dato. Se però, viceversa, sono frequenti le richieste relative agli elementi multipli, e la loro relazione è importante, conviene copiare il singolo all'interno di detti elementi, per evitarne il recupero ogni volta.

Nel caso molti a molti bisogna considerare ancora di più la proporzione delle richieste. Le relazioni molti a molti vengono generalmente descritte da un terzo oggetto, che oltre a mantenere i riferimenti alle due entità, descrive le proprietà della relazione. Le scelte principali che si possono fare in questo caso sono due. Se la lettura è sbilanciata verso uno dei due elementi della relazione, e, allo stesso tempo, è importante che vengano restituiti gli oggetti che descrivono l'associazione assieme all'elemento stesso, è bene integrare le associazioni in quell'elemento. Altrimenti, in caso la necessità di lettura sia equiparabile da entrambe le parti, è necessario mantenere l'associazione come documento indipendente, eventualmente copiando i dati richiesti per evitare di dover recuperare il terzo elemento della relazione.

### 1.1.4 Analisi del dominio

Il dominio descrive i componenti dell'applicazione e le loro relazioni. Ne vengono espresse le dipendenze, i rapporti reciproci e la cardinalità delle relazioni. La sua analisi, integrata con la previsione del carico delle richieste, permette di fornire un quadro dettagliato sulle necessità relative, per poter poi definire la tipologia delle strutture in cui salvare i dati e le caratteristiche richieste al database.

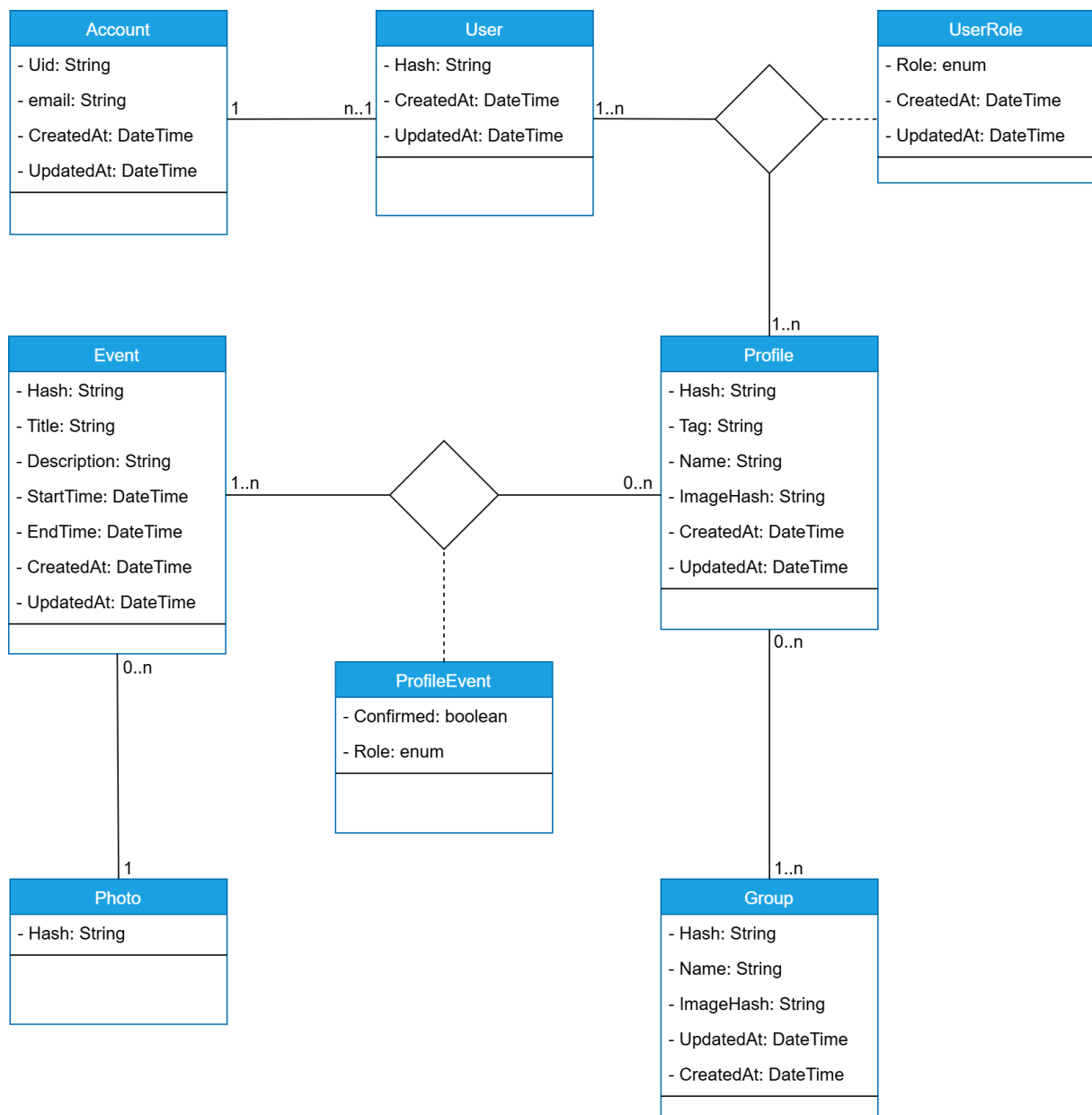


Figura 1.1: Diagramma del dominio di Wyd

Le entità Account, User e Profile, assieme alla relazione UserRole, descrivono le specifiche di autenticazione, identificazione dell'utente e i suoi ruoli sui profili associati. I loro dati vengono recuperati solamente una volta nel ciclo di vita del programma, a seguito del login dell'utente. Vengono poi salvati in memoria locale, riducendo il numero di richieste relative successive a un controllo su eventuali aggiornamenti. Si prevede che le modifiche a questi elementi siano sporadiche.

Allo stesso modo gli oggetti Group vengono recuperati solo all'avvio dell'applicazione, e, salvo rari aggiornamenti, non comportano ulteriori richieste. Le richieste di lettura e scrittura previste per questi elementi sono quindi in quantità esigua, e influiscono secondariamente sulle performance del sistema.

La maggioranza delle richieste verterà sull'ottenimento dei dati relativi agli Event e ai Profile. Gli Event descrivono gli impegni ai quali i profili partecipano. Essendo la loro relazione di tipologia molti a molti, si introduce l'entità ProfileEvent che descrive l'associazione evento-profilo, indicando che un evento è condiviso con un profilo. La proprietà più importante di ProfileEvent è sicuramente Confirmed, una variabile booleana che esprime la partecipazione o meno di un profilo all'evento. È ragionevole prevedere che la cardinalità dei profili associati a un evento non superi l'ordine delle centinaia. Agli eventi che si associano ai profili l'ordine di grandezza invece previsto risiede nelle migliaia. Ci sono molteplici situazioni da tenere in considerazione analizzando questa relazione.

Sicuramente bisogna considerare il recupero specifico delle entità Event e Profile. Questo accade quando un utente decide di entrare nel dettaglio di uno dei due elementi, comportando la richiesta di tutti i dati dell'oggetto. Consistono in un'operazione di lettura tutto sommato semplice: essendo l'utente in possesso dell'identificativo dell'elemento, la ricerca risulta diretta, e i dati da recuperare esigui.

La conferma o la disdetta a un evento vede la modifica di un ProfileEvent. Queste operazioni di per sé risulta veloce in quanto necessita di recuperare un solo elemento per poi modificarlo. Allo stesso modo la modifica di un evento comporta l'aggiornamento di un solo oggetto Event, a seguito però del recupero del ProfileEvent associato, necessario per controllare i permessi del profilo sull'evento. Entrambe le operazioni richiedono però

l'invio di una notifica verso tutti i profili interessati, che necessita il recupero di tutti i ProfileEvent associati ogni volta che una di queste accade.

La probabilità che un evento o un qualunque ProfileEvent a esso associato venga modificato risulta quindi complessivamente elevata e, di conseguenza, l'operazione di recupero degli identificativi di tutti i profili associati all'evento è da considerarsi frequente. Questa operazione, in base all'implementazione, può risultare costosa: una separazione, un accoppiamento debole o la mancanza di indici tra Event e ProfileEvent determinano la necessità di ricercare le entità in tutto il database.

Quando un utente accede per la prima volta su un dispositivo è necessario ottenere gli eventi associati ai suoi profili. Allo stesso modo, a ogni avvio dell'applicazione si recuperano gli eventi che hanno subito modifiche dall'ultimo accesso. Inoltre, per poter garantire (all'interno di un determinato vincolo temporale) la consistenza dei dati anche a livello locale, il client applica una strategia di long polling per ottenere gli eventi che sono stati modificati dall'ultimo aggiornamento noto. L'operazione di ritrovamento degli eventi a partire dal profilo risulta quindi centrale e frequente, per quanto possa accettare un tempo di esecuzione leggermente più lungo.

<b>Funzionalità</b>	<b>Frequenza</b>	<b>Complessità</b>
Recupero di un Event	Media	Semplice
Recupero di un Profile	Media	Semplice
Modifica di un Event	Media	Semplice
Conferma/disdetta di un Event	Alta	Semplice
Ritrovamento dei Profile associati a un Event	Alta	Complessa
Ritrovamento degli Event associati a un Profile	Alta	Complessa

Tabella 1.1: Funzionalità principali tra Event e Profile

La relazione tra Event e Photo impatta sulle prestazioni del sistema solo in casi particolari e verrà affrontato nei capitoli successivi.



## 1.2 L'implementazione del database

Nelle sezioni precedenti si è discusso delle proprietà offerte dai diversi tipi di database e delle necessità che il dominio impone sul sistema. La scelta del database deriva quindi dall'incrocio di tutte queste condizioni, individuando la tecnologia che meglio riesce a rispondere alle esigenze del progetto. Ogni tipologia di database comporta un approccio differente alle informazioni, implicando una strategia di salvataggio e manipolazione dei dati propria. Le strutture che modellano le entità devono quindi essere create per sfruttare nella maniera più efficiente possibile i vantaggi offerti dalla tecnologia scelta.

Una volta scelto il database e le strutture in base alla modalità che più si addicono alle esigenze del progetto, l'utilizzo di servizi in cloud comporta una maggiore attenzione anche alle proprietà legate al mantenimento del servizio, dalle quali derivano le proprietà di scalabilità e affidabilità. La grande differenza tra i vari servizi sta nelle proprietà del server incaricato di fornire il potere computazionale necessario per l'esecuzione. L'architettura del server e la sua integrazione con la tecnologia del database determinano infatti l'effettiva capacità di scalabilità del servizio.

Si intende scalabilità verticale la capacità di aumentare le risorse della stessa macchina in cui si esegue il codice. La scalabilità verticale viene definita nel momento di creazione del servizio, in cui si determinano le risorse da dedicare alla macchina che esegue il programma. Trattandosi di macchine virtualizzate, è sempre possibile in un secondo momento aumentare le prestazioni in caso di necessità.

Per scalabilità orizzontale si intende invece la capacità di delegare il carico di lavoro ad altre macchine, eventualmente coordinando le modifiche. Questo permette una risposta alle richieste più resistente, riducendo il rischio di colli di bottiglia che potrebbero venirsi a formare nell'utilizzo di un nodo singolo. La scalabilità orizzontale richiede però l'implementazione di tecnologie apposite integrate con il database che permettano l'esecuzione in nodi fisici differenti.

Una volta individuata la tecnologia adatta e il livello di scalabilità desiderati, è bene considerare le altre necessità o le opportunità aggiuntive generate dalla presenza di un

database nel progetto.

L’alta disponibilità(HA) è la proprietà di garantire l’accesso al servizio nonostante i guasti. Ad esempio, si può mantenere una macchina identica al server principale in grado di replicare il servizio, spostando il carico in caso di guasto del server principale. Si misura in “numero di nove”, ovvero la quantità di nove presenti nella percentuale del tempo per il quale si garantisce la disponibilità del servizio. I servizi offrono diverse qualità di HA, in base alle funzionalità desiderate.

Alcuni servizi possono presentare offerte di backup per riportare il server nello stesso stato di qualche momento precedente. Questo permette il ripristino del sistema a un punto precedente rispetto all’avvenimento di eventuali errori o guasti del sistema.

### **1.2.1 La scelta del database**

Viste le necessità del progetto in ambito di scalabilità e le caratteristiche del dominio, si individua nei database documentali la tecnologia più adatta per gestire la persistenza centrale dell’applicazione.

I database documentali, facenti parte della categoria dei database non relazionali, rappresentano un paradigma di gestione dei dati che organizza le informazioni in documenti. Ogni documento è un’unità autonoma che incapsula la descrizione di un’entità, contenendo le sue proprietà. Tali documenti sono logicamente raggruppati in collezioni. All’interno di una collezione, ciascun documento è univocamente identificato da un proprio identificativo, garantendo l’accesso diretto e la manipolazione individuale.

Un aspetto distintivo e strategicamente rilevante dei database documentali è la loro intrinseca capacità di supportare la scalabilità orizzontale in modo nativo. Questo è un vantaggio fondamentale in architetture distribuite e ambienti ad alta intensità di dati. La scalabilità è realizzata attraverso la partizionamento (o sharding), un meccanismo che distribuisce automaticamente i dati tra diversi nodi di archiviazione fisici.

Attraverso la denormalizzazione si possono migliorare le prestazioni di lettura, aggregando dati correlati all'interno di un singolo documento o partizione. Questo approccio semplifica la gestione delle join, che possono essere ottimizzate per risiedere all'interno della stessa partizione o in partizioni vicine, minimizzando la necessità di operazioni di lettura tra nodi distinti.

La denormalizzazione comporta intrinsecamente alcune sfide a livello di consistenza dei dati, in particolare per le operazioni di recupero che coinvolgono dati potenzialmente duplicati. Questo problema è efficacemente mitigato dall'adozione di Global Secondary Indexes (GSI). I GSI consentono di interrogare i dati su attributi che non sono la chiave primaria del documento, e risiedono quindi su partizioni differenti. Forniscono percorsi di accesso alternativi e performanti sull'intero dataset distribuito, superando le limitazioni imposte dalla distribuzione fisica delle partizioni e mantenendo un'elevata efficienza nelle query complesse.

Implementando automaticamente e nativamente la scalabilità orizzontale, il database relazionale ci permette quindi di gestire con efficienza l'incremento dei volumi di dati e dei carichi di lavoro senza interventi complessi. Fornisce inoltre un supporto diretto all'esigenza dell'architettura riguardo alla necessità di letture performanti da entrambi i lati di relazioni molti-a-molti: attraverso il partizionamento strategico, i dati correlati possono essere collocati in partizioni vicine per ottimizzare le letture da un lato della relazione, mentre i Global Secondary Indexes (GSI) superano le limitazioni delle partizioni fisiche, consentendo interrogazioni efficienti e performanti dall'altro lato. Infine, un'attenta progettazione del modello di dati, che include una denormalizzazione strategica e l'utilizzo degli indici, garantirà un tempo di recupero ridotto per le informazioni, massimizzando la reattività del sistema e l'efficienza complessiva.

Un confronto con il paradigma relazionale evidenzia le ragioni della sua esclusione per le esigenze del nostro progetto. Sebbene i database relazionali siano soluzioni consolidate per la gestione di dati strutturati, presentano delle limitazioni che non si allineano con i requisiti di scalabilità richiesti. La loro architettura, che spesso lega indici e tabelle alla stessa partizione logica, impone intrinsecamente dei vincoli sulla scalabilità orizzontale,

non solo per la difficoltà di distribuire indici e tabelle su nodi diversi, ma anche per il numero massimo di connessioni contemporanee che possono gestire, limitando la capacità di rispondere a un numero massiccio di richieste simultanee. L'implementazione dello sharding, sebbene possibile, è interamente a carico dello sviluppatore, introducendo un significativo onere di progettazione, sviluppo e manutenzione.

Inoltre, la gestione di relazioni molti-a-molti nel modello relazionale, pur essendo logicamente chiara con tabelle di giunzione, può presentare problemi di scalabilità in contesti di elevato carico. Quando una query necessita di recuperare dati da entrambi i lati di una relazione molti-a-molti, ciò implica l'esecuzione di join complesse tra più tabelle. Sebbene queste operazioni possano essere veloci su singole istanze di database ben ottimizzate, in un ambiente distribuito e con volumi di dati in crescita, queste join possono richiedere il trasferimento di grandi quantità di dati tra nodi diversi per essere risolte, introducendo latenza e overhead di rete significativi. Questo può diventare un collo di bottiglia, compromettendo le performance complessive. Infine, la mancanza di un supporto nativo per indici secondari globali (GSI) rende più complessa la gestione di query su attributi non chiave distribuiti su diverse partizioni, costringendo a soluzioni alternative che potrebbero compromettere la reattività del sistema e aumentare la complessità del codice. Questi fattori combinati ci hanno portato a escludere il modello relazionale.

Essendo il progetto già improntato sulla piattaforma Azure, la ricerca verte inizialmente tra le opzioni che mette a disposizione. Azure offre un'ampia scelta di database documentali che possono essere integrati con il resto dell'ecosistema. Tuttavia, Azure presenta un servizio completamente gestito e nativo per i database non relazionali chiamato Azure Cosmos DB. Garantendo la massima interoperabilità all'interno dell'ecosistema, si procede analizzando le proprietà e i vantaggi offerti da Cosmos DB.

Azure Cosmos DB si distingue per la sua capacità di scalare orizzontalmente in maniera illimitata, consentendo di gestire volumi di dati e carichi di lavoro molto elevati, fino a milioni di richieste al secondo, grazie alla possibilità di distribuire il carico su più regioni Azure. È stato infatti ideato per essere presentare un'architettura distribuita, con replica automatica dei dati, assicurando un'elevatissima disponibilità e resilienza. Queste

## 1 – La gestione della persistenza





















vengono assicurate anche in caso di interruzioni regionali, grazie a meccanismi di failover automatico. Inoltre, la distribuzione globale "turnkey" garantisce che i dati siano sempre vicini agli utenti, riducendo drasticamente la latenza a millisecondi a cifra singola (con SLA del 99.999% di disponibilità per account multi-regione). Consente l'indicizzazione attraverso più partizioni in maniera automatica e personalizzabile ottimizzando le query, riducendo la complessità e migliorando le prestazioni, senza richiedere oneri di gestione manuale degli indici.

Pur essendo focalizzato sui database documentali, Cosmos DB è però una soluzione multi-modello e multi-API. Supporta infatti, oltre alla sua API nativa per NoSQL (che usa il modello a documenti JSON), anche API compatibili con MongoDB, Apache Cassandra, Apache Gremlin (per i grafi) e Azure Table. Questa versatilità permette agli sviluppatori di utilizzare strumenti familiari, semplificando la migrazione di applicazioni esistenti o lo sviluppo di nuove con la flessibilità di scegliere il modello di dati più appropriato.

Pricing : All | Operating System : All | Publisher Type : All | Product Type : All | Publisher name : All

☐ Azure services only

Showing 1 to 20 of 244 results for 'document database'. [Clear search](#)

Name	Publisher	Plan	Price starts at
 <a href="#">Hyperscience Intelligent Document Processing</a>	Hyperscience	Hyperscience	Price varies
 <a href="#">MarkLogic Multi-Model Database - Enterprise Edition v.10</a>	MarkLogic	MarkLogic 10.0-11 Cluster Deploy...	Price varies
 <a href="#">TerminusDB In-Memory Document Graph Database</a>	TerminusDB	TerminusDB Enterprise	€0.283/3 years
 <a href="#">MarkLogic Multi-Model Database - Enterprise Edition v.11</a>	MarkLogic	MarkLogic 11.3.1 Cluster Deployment	Price varies
 <a href="#">DbGate (database manager)</a>	Sprinx Systems, a.s.	DbGate 6	€0.025/hour
 <a href="#">P.AI by Polestar - Gen AI Bot for Insights, Database Query and Document P...</a>	Polestar Solutions & Services India LLP	Requirement specific planning	
 <a href="#">Azure Cosmos DB</a>	Microsoft	Azure Cosmos DB	
 <a href="#">P.AI by Polestar - Gen AI Bot for Insights, Database Query and Document P...</a>	PolestarInsights	Client specific plan	
 <a href="#">Pre-configured DBeaver for Seamless Database Management</a>	TechLatest	Pre-configured DBeaver for Seamless Database Management St	€0.026/hour
 <a href="#">ArangoDB NoSQL Database on Ubuntu 1</a>	Apps4Rent LLC	ArangoDB NoSQL Database on Ubuntu 18.04 LTS	€0.005/hour
 <a href="#">Xpert BI with Azure SQL Database (Azure SQL DB)</a>	BI Builders as	Xpert BI with Azure SQL Database	
 <a href="#">MongoDB Server</a>	Cloud Infrastructure Services	MongoDB on Ubuntu 22.04	€0.028/hour
 <a href="#">MongoDB Server</a>	Cloud Infrastructure Services	MongoDB on Ubuntu 20.04	€0.028/hour
 <a href="#">RavenDB Cloud</a>	Hibernating Rhinos	Pay-As-You-Go	Price varies
 <a href="#">AllegroGraph 7.3.0</a>	Franz Inc.	AllegroGraph Free Edition	€0.253/hour
 <a href="#">MongoDB® 4.4 on LINUX CentOS 8.2</a>	Tidal Media Inc	MongoDB® 4.4 on LINUX CentOS 8.2	€0.03/hour
 <a href="#">MongoDB Server on Linux 7.9</a>	Art Group	MongoDB Server on Linux 7.9	€0.032/hour
 <a href="#">MongoDB 6 on CentOS</a>	AskforCloud LLC	MongoDB 6 on CentOS	€0.042/hour
 <a href="#">NuOCR - OCR automation</a>	Nuvento LLC	Gold	
 <a href="#">MongoDB 6 on Red Hat Enterprise Linux 9</a>	AskforCloud LLC	MongoDB 6 on Red Hat Enterprise Linux 9	€0.035/hour

Previous

Page 1 of 13

Next

Figura 1.2: Proposte di Azure per i database documentali

A livello di costi è difficile fare un'analisi precisa, in quanto tutti i competitor presenta-

no modelli di pagamento che utilizzano metriche di utilizzo diverse, rendendo necessarie ulteriori analisi che dipendono anche dall'effettivo numero e tipologie di richieste che vertono sul database. Di seguito viene riportata una tabella per comparazione i costi delle alternative principali. Cosmos usa come metrica le Request Units(RU) per quantificare l'impatto di una richiesta sul database. Le RU rappresentano un'astrazione delle risorse di sistema (CPU, I/O, memoria) necessarie per eseguire tali operazioni.

<b>Servizio</b>	<b>Costo ogni milione di scritture (normalizzate a 1 KB)</b>	<b>Costo ogni milione di scritture (normalizzate a 1 KB)</b>
AWS DynamoDB	\$1.25	\$0.25
Google Cloud Firestore	\$0.90	\$0.30
Azure Cosmos DB	In base al consumo di RU, \$5.84 al mese ogni 100RU/s	In base al consumo di RU, \$5.84 al mese ogni 100RU/s
MongoDb Atlas	Varia in base all'host	Varia in base all'host

Tabella 1.2: Funzionalità principali tra Event e Profile

Inoltre, Azure mette a disposizione molteplici servizi accessori che possono essere uniti al servizio. Questo permette di estendere le potenzialità del database tramite l'analisi e il monitoraggio dei dati, generando prestazioni aggiuntive o integrando i dati per lo sviluppo di altre tecnologie.

Al server principale è stata affiancata una replica che rimane costantemente aggiornata. Situata in una località differente dal server principale, garantisce alta disponibilità continuando a fornire i servizi anche in caso di malfunzionamenti al server principale.

Nel caso in cui però fossero necessarie ulteriori prestazioni, se il dominio e i requisiti lo permettono, si può eventualmente delegare a un database non relazionale le modifiche ai dati e alle relazioni che non necessitano delle qualità ACID ma richiedono un'alta frequenza di scrittura.

Interporre una cache tra la logica applicativa e il database semplifica e riduce il numero di richieste verso il database. Il livello di caching si occupa di gestire le richieste al database fornendo e duplicando le risposte che possiede già in memoria, eventualmente concentrando le richieste in caso i dati siano invece da recuperare. Per i dati in scrittura, invece, salva temporaneamente le modifiche richieste, aggiornando subito la memoria locale, per poi apportare le modifiche al database in momenti di carico ridotto. Garantisce così un tempo di risposta e di propagazione degli aggiornamenti ridotto, alleviando il numero di richieste al database, estendendo così le prestazioni fornite.

Tuttavia, non vi è alcun vincolo che impedisca l'affiancamento di database di tipologia diversa per rispondere a esigenze specifiche e sfruttare i punti di forza di entrambe le tecnologie.

### 1.2.2 La definizione delle classi

Nonostante sia più probabile che venga richiesto il dettaglio di un evento, e quindi sia più frequente il dover recuperare i Profile relativi all'Event, la proporzione delle richieste prevista non giustifica lo sbilanciamento della relazione sugli Event. Infatti, se si salvassero tutti i ProfileEvent sull'oggetto Event, le richieste degli eventi appartenenti ai profili, sebbene meno frequenti, richiederebbero l'ispezione di tutti gli Event alla ricerca del Profile indicato. Infine, se si duplicasse l'oggetto ProfileEvent, oltre che nella sua tabella

1 – La gestione della persistenza

originaria, anche sugli Event la sua creazione, la sua eliminazione e la modifica del campo Confirmed richiederebbero il doppio delle scritture.

### **1.2.3 L'integrazione con le Azure Functions: il framework .Net e l'eventual consistency**



### 1.2.4 L' integrazione con C#

La scelta di un database relazionale per la persistenza ha comportato sviluppi progettuali precisi. In primis si rende necessario tradurre il dominio in componenti relazionali che possano essere espressi e salvati nelle tabelle del database.

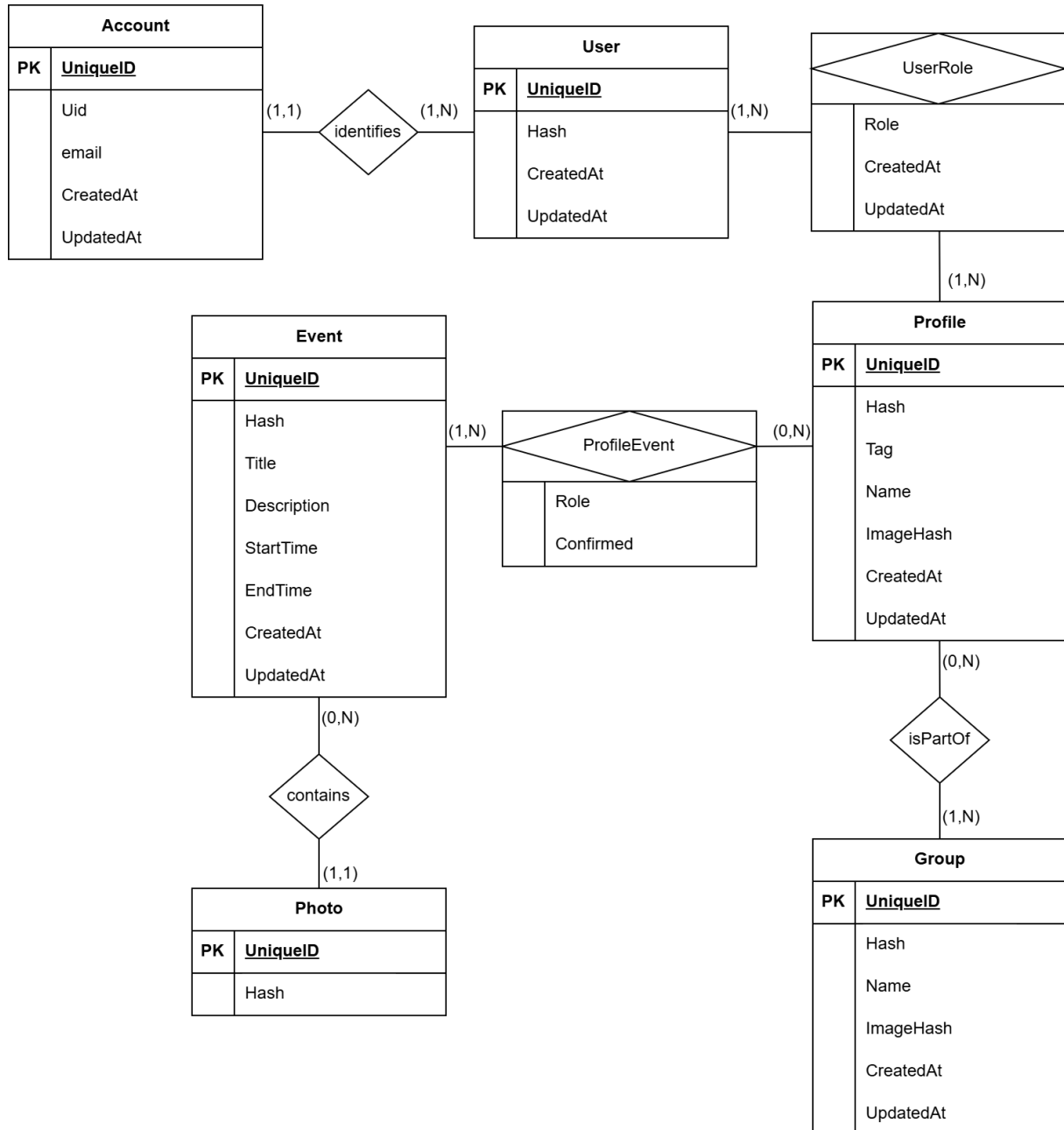


Figura 1.3: Diagramma Entità - Relazione del dominio

Si creano quindi sul server le classi logiche del programma, a partire dal dominio. Ogni classe corrisponde ad un oggetto del dominio, presentando i valori e le relazioni dei componenti come attributi dell'oggetto.

Entity Framework Core di .Net(EFCore) è una libreria di C# che permette di unire le classi logiche del programma alle tabelle del database. Fornisce un’astrazione logica del collegamento con il database e le richieste relative, fornendo una rappresentazione di alto livello delle connessioni sottostanti.

Una volta collegato il server con il database tramite le stringhe di connessione salvate sull’Azure Key Vault, sono state definite le proprietà tra le varie entità, per poi inizializzare in automatico la struttura del database. Le modifiche alla struttura del database vengono infatti generate automaticamente da EFCore in seguito alla creazione o alla modifica degli attributi degli oggetti. Questo permette di star dietro agli aggiornamenti, generando e salvando le modifiche da applicare ad ogni modifica delle proprietà del dominio.

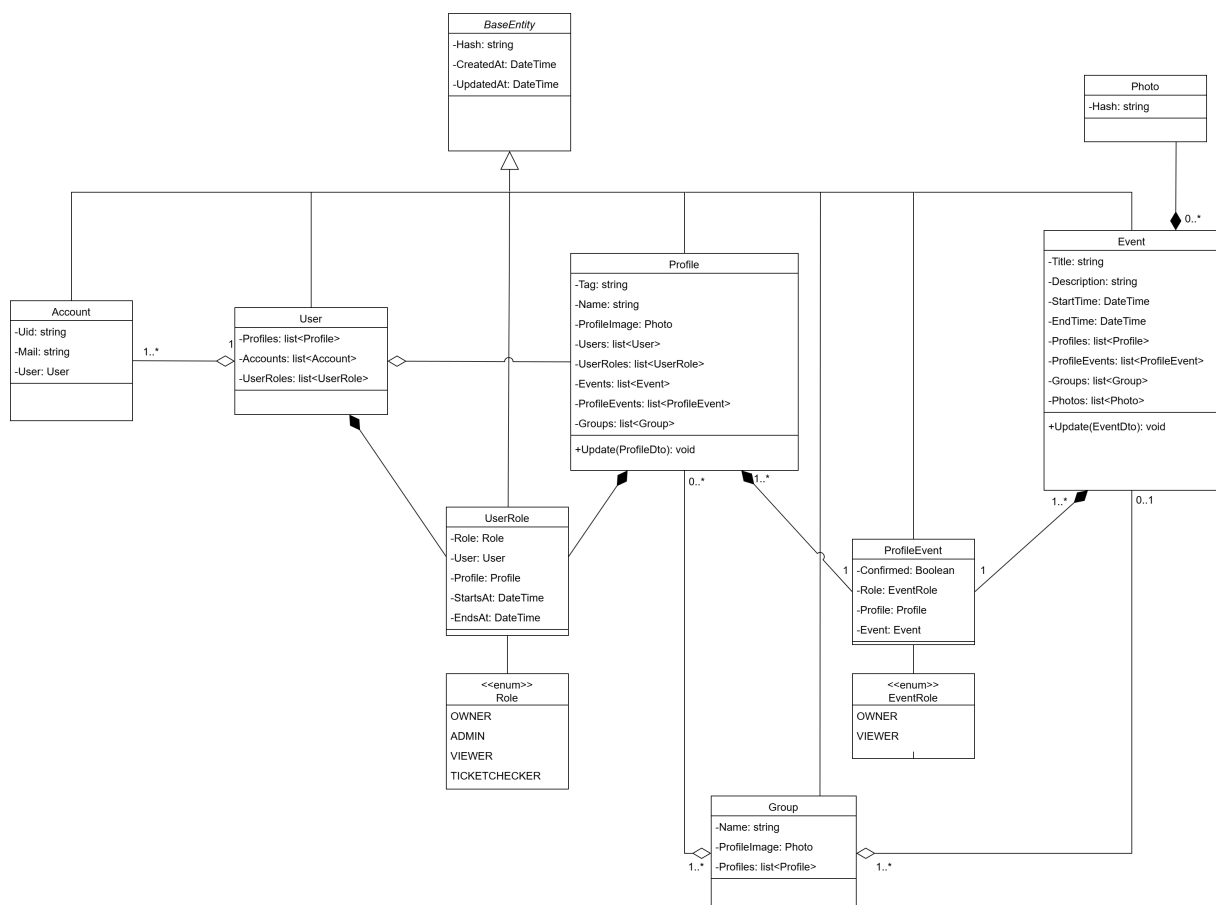


Figura 1.4: Modello delle classi del client

Per la riduzione del carico computazionale richiesto da elementi con tante relazioni si utilizza la tecnica del lazy loading. La tecnica del Lazy Loading consiste nel richiedere i dati delle relazioni di un elemento solo quando strettamente necessario. La sua realizzazione tramite EFCore è attuata grazie alla proprietà virtual, che permette di gestire un oggetto con un riferimento al database richiedendo i dati delle sue relazioni solo quando viene espressamente richiesto.