Alma Mater Studiorum · Università di Bologna

Campus di Cesena
Corso di Laurea in Ingegneria e Scienze Informatiche

Studio dello standard GraphQL nella Model Driven Programming: Analisi di un prototipo di generatore di codece e scrittora di una versione per librerie ORM SQL

Tesi di Lurea in Programmazione ad Oggetti

Relatore:
Prof. Mirko Viroli

Presentata da:
Ivan Perazzini

Terza Sessione di Laurea
Anno Accademico 2019/2020

Introduzione

Gli ultimi anni hanno visto una maturazione dello standard[1] GraphQL, dei servizi costruiti per implementarlo e la sua adozione da parte di numerose aziende[2]. Definito per la prima volta internamente a Facebook nel 2012, oggi si presenta come uno standard capace di competere con l'architettura di tipo REST per il dialogo fra client e server.

Caratterista fondamentale di GraphQL è la possibilità di definire uno schema che permette di modellare il formato delle query esposte e quello delle entità restituite al client, di modo che questo possa effettuare richieste personalizzate, ad un singolo endpoint e vedersi restituire dati già tipizzati. Le entità sono definite da campi tipizzati e relazioni con altre entità, la struttura del dominio delle interrogazioni gestibili dal server è strutturata come un grafo.

Le possibilità offerte dallo standard però non si limitano alla comunicazione fra client e server: essendo i campi delle entità tipizzati, ed avendo lo standard un concetto di introspection, è possibile usare schemi GraphQL anche per definire lo strato del model. Questa possibilità rende quindi l'impiego di questo standard potenzialmente adatto per lo sviluppo Model Driven, potendo generare in modo automatizzato il codice applicativo che si occuperà di interfacciarsi con le strutture definite nello schema GraphQL.

Da qualche tempo, l'azienda Twinlogix, ha deciso di sfruttare questa potenzialità per realizzare un generatore di codice che, basandosi sullo schema, costruisca definizioni di modelli per database. Per poter interagire con questi modelli, il generatore deve anche generare i DAO necessari. A tal scopo, l'azienda sta anche lavorando alla scrittura di un DAO generico e metodi per interagirci indipendenti dal tipo dell'oggetto richiesto e dal database sottostante. È stato già scritto un primo prototipo di questo DAO e del generatore indirizzato a database MongoDB. Obiettivi dell'attività di seguito discussa sono stati quindi:

- studiare le tecnologie necessarie per lo svolgimento dell'attività;
- analizzare l'architettura del DAO e i prototipi sviluppati dall'azienda;

[1] http://spec.graphql.org/

[2] https://landscape.graphql.org/card-mode?category=graph-ql-adopter&grouping=category

- progettare e implementare una prima versione versione del DAO generico e del generatore che, sfruttando librerie ORM, potesse interfacciarsi con database SQL.

Indice

Elenco delle figure

Capitolo 1

Contesto e Tecnologie

Model Driven Programming

Lo sviluppo "Model Driven" è un approccio all'ingegneria del software orientato alla definizione di modelli piuttosto che di software.

La complessità di un progetto può essere catalogata in due macro gruppi:

- *Essenziale*, ovvero derivata dal problema che si cerca di risolvere, indipendente dalle tecnologie impiegate;
- *Accidentale*, dovuta agli strumenti impiegati per affrontare il problema.

Uno degli obiettivi di una buona progettazione è quello quindi di ridurre il più possibile la complessità accidentale. La complessità di molti sistemi software moderni è talmente elevata da renderne difficile sia la comprensione (basandosi solo sul codice scritto), sia la realizzazione. Parlare di modelli permette di rendere comprensibile l'architettura di un progetto e di potersi concentrare sugli obiettivi, dimenticandosi delle complicazioni e limitazioni che verrebbero incontrate in fase di sviluppo.

La programmazione in sé è un'attività con un'alta potenzialità di errori accidentali: una sola riga di codice sbagliata è infatti tutto quello che basta per causare costosi danni [1].

[1] http://users.csc.calpoly.edu/ jdalbey/SWE/Papers/att_collapse

Spesso, in fase di implementazione di sistemi complessi, ci si trova poi davanti a blocchi di codice strutturati in modo estremamente ripetitivo e complesso: questa è una situazione nella quale errori accidentali sono quasi assicurati. In questo tipo di sezioni la modellazione può incontrare facilmente la generazione di codice: questi blocchi di codice possono essere descritti in modo astratto ed indipendente da linguaggi di programmazione e librerie.

Utilizzare questo approccio dove utile porta a notevoli benefici:

- il tempo e le forze degli ingegneri sono utilizzati in modo migliore: tutta la complessità dell'implementazione è spostata sulla scrittura dei generatori di codice, piuttosto che di sezioni ripetitive;
- un progetto strutturato con modelli diventa più comprensibile anche per il cliente;
- in caso siano necessarie modifiche, basterà effettuarle al modello. I cambiamenti sono poi generati rapidamente e senza rischio di errori accidentali.

ORM

L'*Object Relational Mapping* è una tecnica di programmazione creata per favorire l'interazione fra un'applicazione scritta con un approccio ad oggetti ed una base di dati di tipo relazionale, o comunque modellante tabelle tramite campi scalari semplici. Una libreria ORM rende quindi possibile mappare le tabelle in oggetti e viceversa, permettendo di poter operare con questi dentro l'applicazione anche quando si deve comunicare con un DB. Esempio di confronto fra l'utilizzo di un driver DB e quello di una libreria ORM (fonte snippets: en.wikipedia.org):

```
{
  var sql = "SELECT * FROM persons WHERE id = 10";
  var result = context.Persons.FromSqlRaw(sql).ToList();
  var name = result[0]["first_name"];
}

{
  var person = Person.Get(Person.Properties.Id == 10);
  var firstName = person.GetFirstName();
}
```

Gli immediati vantaggi di questo approccio sono il ridotto numero di righe di codice scritte e la conseguente pulizia del codice, nonché la semplicità nell'interazione con il DB e la minor probabilità di errore, non dovendo scrivere esplicitamente interrogazioni in linguaggio SQL. D'altro canto, utilizzare una libreria ORM ha anche numerosi svantaggi:

- E' una libreria complessa se la si vuole sfruttare a fondo
- Toglie del controllo al programmatore vista la grande astrazione che introduce
- Può produrre query sub-ottimali

Questi svantaggi non sono però stati ritenuti abbastanza influenti da decidere di optare per una soluzione di più basso livello.

GraphQL

Ideato da Facebook nel 2012, il linguaggio GraphQL è stato successivamente pubblicato in maniera open source e, dal 2018, è gestito come standard dalla GraphQL Foundation. Nasce con il duplice scopo di modellare entità collegate fra di loro da relazioni e di fornire una soluzione a noti problemi dell'approccio RESTful, architettura per la comunicazione client server prevalente nello sviluppo in ambito web. Il linguaggio di GraphQL permette di definire uno schema costituito dai tipi:

Oggetto Nodi che rappresentano la struttura delle entità gestite dall'applicativo.

```
type Address {
  street: String
  city: String
  country(continent: Continent): Country!
}
```

Esempio di tipo oggetto: è definito un oggetto nominato "Address" con tre proprietà tipizzate. Il nodo contiene due valori scalari ("street" e "city"), mentre la proprietà "country", di tipo oggetto; questo indica che esiste un collegamento fra il nodo Address e quello Country. Si nota anche come le proprietà possano ricevere dei parametri in input.

Query e mutazioni richieste che il client può effettuare al server, rappresentano i punti d'ingresso (dopo il nodo root, implicitamente usato) nel grafo. Le *query* rappresentano operazioni di lettura, mentre le *mutations* quelle di modifica.

```
schema {
  query: Query
  mutation: Mutation
}
```

Definizione delle keyword *query* e *mutation*.

```
type Query {
  address(continent: Continent): Character
  user(id: ID!): User
}
```

Esempio di definizione delle query esposte.

Quando un client contatta un server GraphQL, può fornire come parametri di richiesta una operazione di query o mutazione fra quelle definite nello schema specificandone la forma dell'output. Le immagini successive mostrano un esempio di comunicazione con un server GraphQL in contrasto ad uno che implementa l'API REST.

Figura 1.1: Esempio di comunicazione con server REST.

Fonte: www.howtographql.com

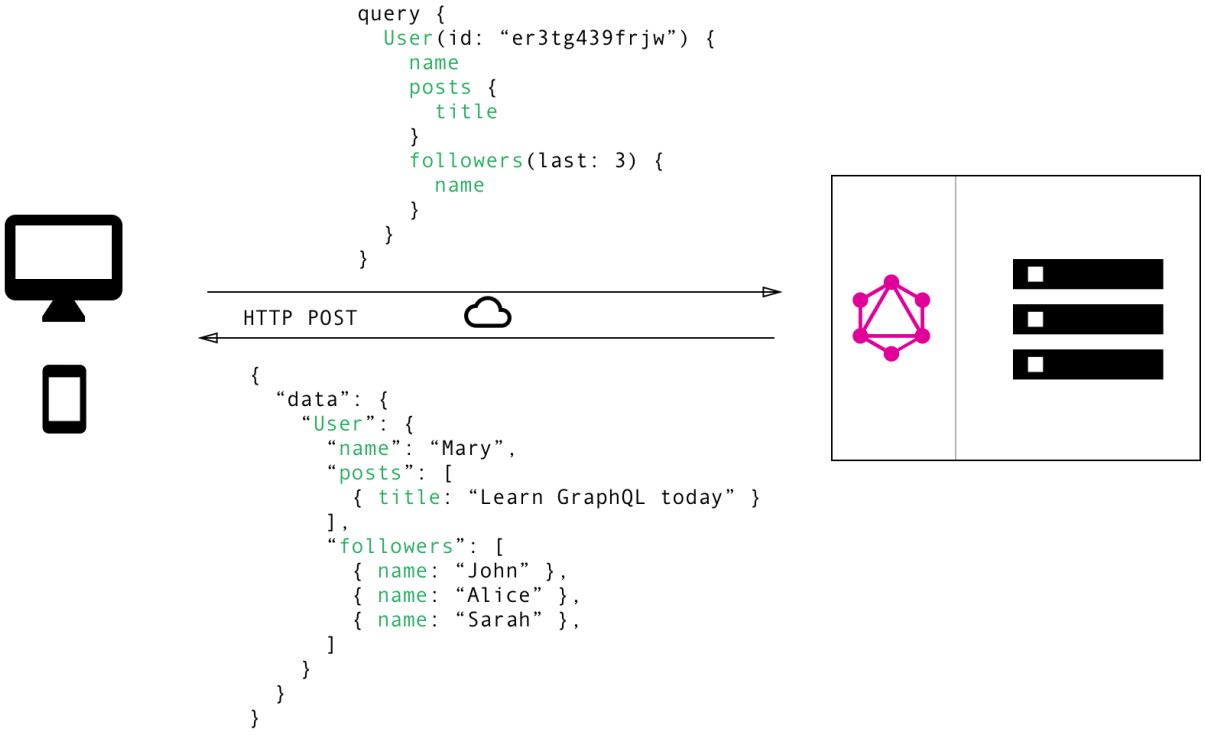

Figura 1.2: Esempio di comunicazione con server GraphQL.

Fonte: www.howtographql.com

Si nota quindi che l'utilizzo di GraphQL permette di risolvere alcuni problemi dell'approccio REST:

Overfetching e underfetching Ovvero il fatto che, essendo il formato delle risposte del server fisso, il client spesso riceve più (o meno) informazioni di quelle necessarie. Nel caso in cui ne riceva meno, la cosa comporta anche la necessità di effettuare altre richieste al server per ricevere i dati necessari. Come si può notare dalle immagini di esempio, in una richiesta GraphQL è il client a specificare esattamente il formato della risposta.

Multipli endpoint Lo schema GraphQL indica un solo punto d'ingresso, il nodo root; da quello si può accedere alle varie query e mutazioni. Il fatto che ci sia un solo punto d'ingresso significa che per il client basterà contattare un unico endpoint per ricevere tutte le informazioni di cui ha bisogno. Questo è evidenziato nell'immagine di esempio di comunicazione con server REST: il client ha dovuto contattare tre endpoint differenti.

Modifiche all'API Il fatto che in un'architettura REST sia il server a dover definire il formato dei risultati di una chiamata comporta che l'API di back-end debba essere modificata spesso per riflettere i cambiamenti nell'interfaccia front-end, se non si vuole incorrere in fenomeni di over o underfetching. Lo standard GraphQL permette quindi una più rapida produzione di applicazioni e un'API back-end più stabile.

Interpretazione dell'output Essendo i campi dei nodi GraphQL tipizzati, i tipi della risposta di un server GraphQL sono inequivocabili: questo facilita la comprensione dell'output di una richiesta. Lo standard GraphQL supporta anche il concetto di *introspection*: questo permette, anche senza avere accesso allo schema, di scoprire le query e i formati dei dati. Una soluzione REST prevede invece la scrittura e il mantenimento di una documentazione a riguardo.

Oltre a fornire uno standard robusto per la comunicazione client server, GraphQL può essere usato come linguaggio per definire modelli: sono infatti già disponibili servizi di generazione di codice basata su schemi GraphQL. Quello utilizzato in questo caso è *GraphQL Code Generator*[2].

[2]https://graphql-code-generator.com/

GraphQL code generator

GraphQL code generator è un tool CLI open source in grado di analizzare uno schema GraphQL e, tramite plugins, generare codice su diversi file. Il modo in cui i plugin di questo strumento operano segue il pattern di programmazione *visitor*.

Il pattern visitor è una soluzione al problema che si verifica quando bisogna effettuare operazioni su una struttura di oggetti senza però aggiungere funzionalità agli stessi: ogni oggetto della struttura implementa un'interfaccia che definisce un metodo *accept* che riceve come parametro un'implementazione dell'interfaccia Visitor. Questa specifica invece un metodo *visit* per ogni tipologia di oggetto (ad esempio, ogni classe) sul quale svolgere operazioni.

Nell'applicazione principale poi, per ogni oggetto, viene chiamato il metodo accept passando ogni implementazione di Visitor, cosicché per aggiungere operazioni/trasformazioni delle classi è possibile semplicemente implementare un nuovo Visitor.

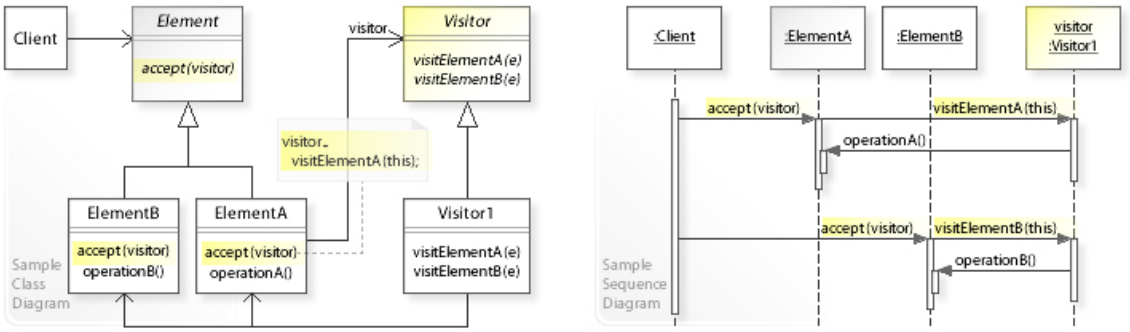

Figura 1.3: Esempio di struttura e funzionamento del pattern Visitor.

Fonte: en.wikipedia.org

È questo il meccanismo alla base di GraphQL code generator: al suo interno, ogni plugin può accedere allo schema del grafo e chiamare il metodo *visit*, passando come parametro un oggetto che funga da visitor. Sono disponibili dei plugin predefiniti, fra cui quello, usato durante questo progetto, per generare tipi, interfacce etc... in linguaggio TypeScript.

TypeScript

TypeScript è un linguaggio open source ideato per arricchire il linguaggio JavaScript aggiungendo i tipi statici. L'uso dei tipi permette la validazione del codice e, di conseguenza, uno sviluppo più fluido.

Oltre a definire tipi di base come *number* o *string*, TypeScript supporta anche tipi composti. Ad esempio:

```
type NetworkFromCachedState = {
  state: "from_cache";
  id: string;
  response: NetworkSuccessState["response"];
};
```

rappresenta un tipo con tre attributi:

- state: è di tipo "from_cache", ovvero un tipo che accetta solo quel valore. Tipi definiti in questo modo sono detti *literal types*
- id: campo di tipo *string*
- response: campo di tipo *typeof NetworkSuccessState["response"]*. Il suo tipo corrisponde a quello di un oggetto definito in un altro tipo.

È possibile creare tipi avanzati, ma la questione non verrà discussa in questo testo.

I tipi, a differenza delle classi, non hanno un operatore nativo (come *typeof*) per distinguerli. Per controllare il tipo di una variabile è necessario definire *type guards* personalizzate.

```
type stringWrapper = {
  value : string;
}
function isString(x: any): x is stringWrapper {
  return typeof x["value"] === "string";
}
function printIfString(x : any) {
  if(isString(x)) {
    console.log(x.value);
  }
}
```

In questo caso, senza la type guard, cercare di usare la proprietà "x.value" genererebbe un errore di validazione del codice, essendo x di tipo any.

Sequelize

In accordo con l'azienda, la versione SQL è stata scritta sfruttando la libreria ORM per ambiente Node *Sequelize* per l'interazione con la base di dati. Sequelize supporta i seguenti RDBMS:

- Postgres
- MySQL
- MariaDB
- SQLite
- Microsoft SQL Server

Descrivere nello specifico il funzionamento della libreria non rientra negli obiettivi di questo documento (alcune funzionalità verranno esposte in seguito) ma, per la comprensione del testo, si introducono di seguito i concetti di *associazione* e *Model* di Sequelize.

Model I Model sono le entità sulle quali la libreria opera. Sono oggetti che astraggono le tabelle salvate sul database. Al loro interno posseggono informazioni sui campi (nome, valore, tipo...), sulla tabella (nome della tabella, nome del model, associazioni...) etc...

Mentre una classe statica relativa ad una tabella rappresenta tutta la tabella ed espone metodi per effettuare modifiche e letture in blocco su di essa, un *istanza* di Model rappresenta una riga di una tabella. I metodi di modifica riguarderanno solo il record associato a quella istanza.

Associazioni Le associazioni sono oggetti usati per gestire le associazioni *uno a uno*, *uno a molti* e *molti a molti* fra tabelle. Esistono quattro tipi di associazioni Sequelize:

- HasOne
- BelongsTo
- HasMany
- BelongsToMany

Per aggiungere associazione con altre tabelle, i model possono invocare metodi chiamati allo stesso modo. Un esempio:

```
//Definizione models
const A = sequelize.define('A', /* ... */);
const B = sequelize.define('B', /* ... */);
A.hasOne(B);
A.belongsTo(B);
A.hasMany(B);
A.belongsToMany(B, { through: 'C' });
```

I metodi di tipo *has* aggiungono una associazione nel model invocante il metodo (*source*) e un attributo che fungerà da *foreign key* nella model passato come parametro (*target*). I metodi di tipo *belongsTo* invece aggiungono sia l'associazione che la foreign key nel target.

Capitolo 2

Requisiti

Requisiti funzionali

- il sistema deve potersi interfacciare con un database SQL, traducendo le query dal formato aziendale a SQL.
- il sistema sviluppato deve essere in grado di tradurre uno schema GraphQL in formato tabellare, in particolare:
  - deve poter generare tabelle SQL ed opportune relazioni basate sui tipi definiti in uno schema GraphQL, particolare attenzione viene posta su tipi dalle caratteristiche più complesse, quali:
    - tipi con composizioni, nel caso un tipo GraphQL ne comprenda altri fra i suoi campi;
    - tipi associati ad altri tipi tramite direttive sullo schema GraphQL;
    - ereditarietà fra interfacce e tipi che le implementano.
  - deve poter generare DAO per gestire la lettura/scrittura sulle tabelle generate;
  - deve poter generare tipi di supporto, come filtri, proiezioni e ordinamenti, personalizzati per ogni DAO;
- il sistema deve poter supportare tipi di dato personalizzati nei campi dei tipi;

Requisiti non funzionali

- la comunicazione con il database e la traduzione delle query devono essere implementate utilizzando la libreria Sequelize;
- l'interazione con i DAO deve essere quanto più vicina a quella della versione per MongoDB, per non dover modificare la parte comune della libreria;
- lo schema GraphQL per generare il codice deve essere quanto più simile a quello per la versione MongoDB. Si possono invece modificare liberamente le direttive aggiuntive usate nello schema;
- il sistema non deve creare un overhead eccessivo per la risoluzione delle operazioni.

Capitolo 3

Architettura del prototipo

L'interazione con lo strato del database è gestita tramite il pattern di programmazione Data Access Object (DAO). Il pattern DAO è una strategia impiegata per separare lo strato funzionale di un'applicazione da quello di persistenza dei dati: permette quindi di sviluppare il lato applicativo senza avere conoscenza di come i dati vengano effettivamente gestiti, così come poter modificare la gestione dei dati senza dover effettuare modifiche al lato applicativo, a patto che le interfacce rimangano inalterate.

Il pattern prevede la definizione di un'interfaccia di alto livello che definisce i metodi di lettura e scrittura su un particolare tipo di oggetto, l'implementazione di questa e dell'oggetto, detto "Data Transfer Object", che non è altro che un'entità sulla quale mappare i dati custoditi nel DataBase.

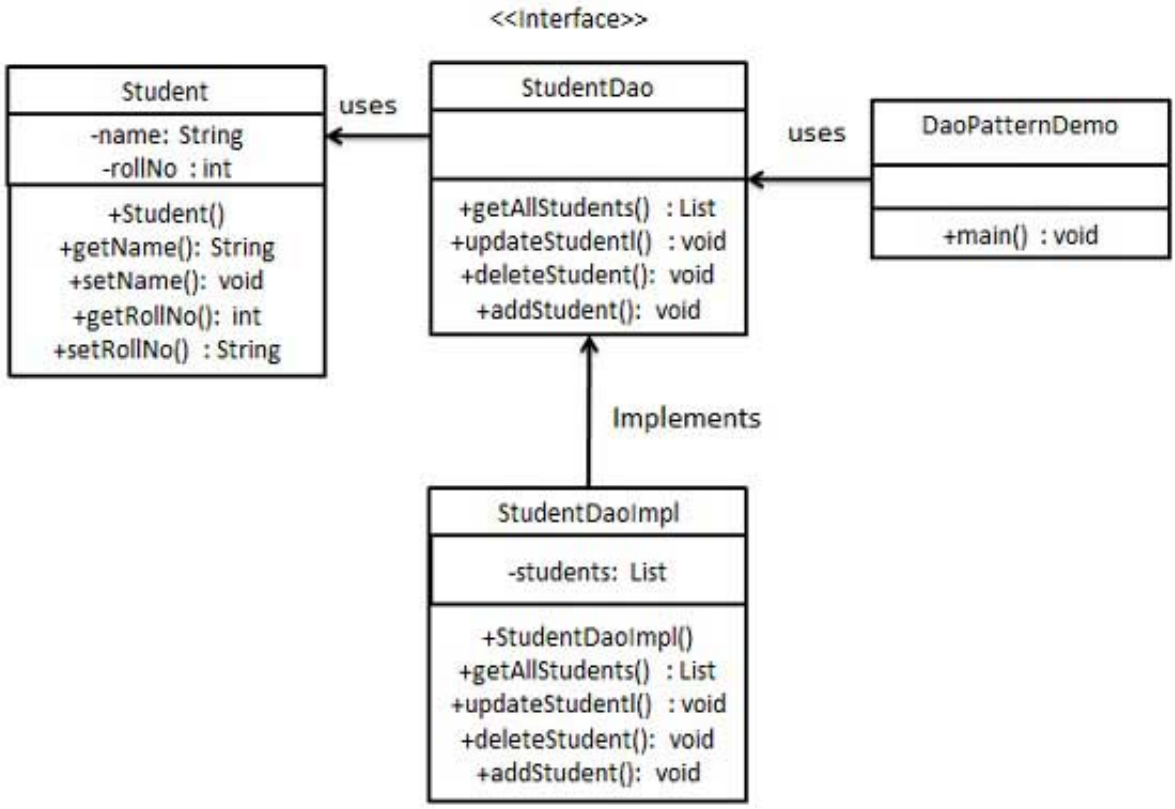

Figura 3.1: Esempio di pattern DAO.

Fonte: www.tutorialspoint.com

L'esempio sopra mostrato descrive un caso nel quale l'applicazione *DaoPatternDemo* utilizza un oggetto implementante l'interfaccia *StudentDao* per ottenere oggetti (DTO) di tipo *Studenti*. Come il dao recupererà i dati non è rilevante per DaoPatterDemo: si è quindi liberi di modificare l'implementazione di StudentDao.

Il progetto però punta a generare dei DAO per oggetti che verranno a loro volta generati in futuro: non è quindi possibile definire DTO e tanto meno interfacce e metodi personalizzati per gestirli. Si potrebbero supportare solo operazioni estremamente generiche, come "elenca tutte i record", "elimina tutti i record", "estrai l'n-esimo record"....

Per superare questa limitazione, l'azienda ha deciso di sfruttare il supporto che Typescript offre per i tipi generici: le interfacce della libreria puntano a definire un DAO

generico adatto ad ogni tipo di DTO.

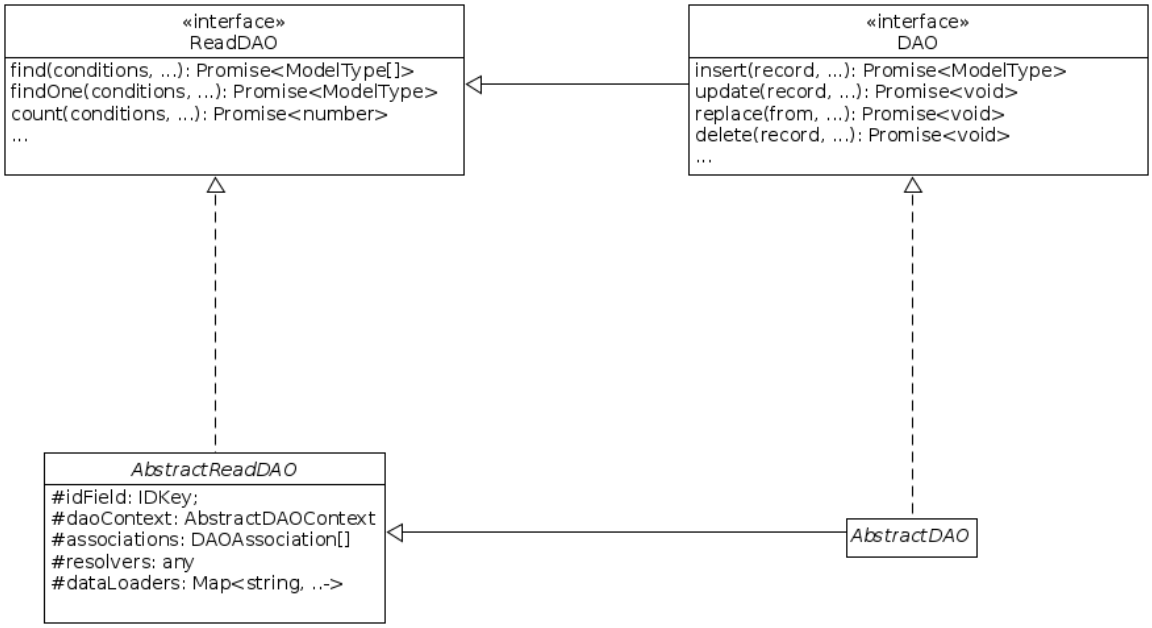

Figura 3.2: Architettura dei principali componenti del prototipo aziendale.

L'entità DAO alla base della libreria è l'interfaccia *ReadDAO*: oltre a definire metodi per la ricerca, l'interfaccia specifica anche le firme per quelli che servono a gestire le *associazioni*.

Viene infatti definito un meccanismo di associazioni tra DAO indipendente dal database sottostante, così da poter implementare associazioni fra i vari tipi dello schema GraphQL anche in database non relazionali.

Le associazioni, oltre all'implementazione dei metodi per gestirle, sono contenute nel classe astratta *AbstractReadDAO*. I metodi di lettura invece sono implementati come template methods.

La struttura principale delle associazioni fra DAO è la seguente:

```
type DAOAssociation = {
  type: DAOAssociationType,
  reference: DAOAssociationReference,
  field: string,
  refFrom: string,
  refTo: string,
  dao: string,
  ...
}
```

- I campi *type* e *reference* indicano se l'associazione è di tipo *foreign/inner* e *uno a molti/uno a uno*;
- *field* indica il nome del campo che andrà risolto come associazione
- *refFrom* e refTo sono i nomi dei campi che fungono da chiavi per la relazione
- *dao* è il nome del DAO associato

Quando un campo con nome uguale al parametro *field* viene ricercato nel DAO che contiene l'associazione, prima viene risolta l'operazione di ricerca sul DAO interrogato, dopodiché viene effettuata una lettura anche sul DAO associato al campo *field* tramite il metodo *load*, definito (ma non mostrato nello schema) nell'interfaccia ReadDAO.

L'interfaccia *DAO* estende l'interfaccia ReadDAO e completa il set di operazioni aggiungendo la definizione dei metodi di scrittura (insert, update, replace...). La sua implementazione astratta si limita a definire questi metodi come template methods.

I parametri dei metodi di lettura e scrittura sono principalmente tre: *filtri*, *ordinamenti* e *proiezioni*. Poiché il DAO deve operare su qualsiasi tipo di DTO, la struttura di questi tre parametri verrà generata durante il processo di generazione del codice e i loro tipi verranno passati fra i parametri generici del DAO generato.

La classe *AbstractDAO* dovrà essere poi implementata da classi astratte dedicate per ogni tipo di DB sottostante: l'azienda ha già definito ed implementato la classe *AbstractMongooseDAO* per gestire la comunicazione con un database MongoDB.

Durante il processo di generazione del codice vengono quindi generate varie implementazioni di questa classe astratta, una per ogni DTO da gestire.

Ultimo componente della libreria è la classe *AbstractDAOContext*. Questa si occupa principalmente di contenere istanze di tutti i DAO dell'applicazione e di inizializzarli. Come mostrato precedentemente nello schema, ogni DAO possiede un riferimento al DAOContext, così che, tramite questo, ogni DAO possa comunicare con gli altri (per esempio per risolvere associazioni).

Capitolo 4

Sviluppo del DAO

Le problematiche maggiori della fase di implementazione del DAO astratto sono scaturite dalle differenze fra database di tipo documentale (come MongoDB) e database SQL. Gli schemi GraphQL utilizzati dall'azienda per eseguire test, e che sono stati presi come punto di riferimento per lo sviluppo dell'implementazione, presentano alcuni tipi difficilmente convertibili in formato tabellare, questo perché gli schemi GraphQL sono più adatti per definire modelli per database NoSQL.

Differenze tra database SQL e NoSQL

I database NoSQL non sono basati su tabelle rigidamente definite e associate fra di loro: esistono diversi tipi di schemi per database NoSQL, fra cui quelli *documentali*, a *chiave-valore* e basati su *grafi*.

In particolare, i database documentali contengono al loro interno *documenti*, strutture dati tipicamente internamente organizzate come serie di chiavi-valori (es: JSON, XML o YAML). A differenza dei database relazionali, tutte le informazioni relative ad un documento sono all'interno della sua istanza e non distribuite fra più tabelle collegate tramite *relazioni*. I database documentali (NoSQL, in generale) contengono principalmente dati di tipo non strutturato e i singoli record possono anche omettere completamente campi descritti nel loro schema.

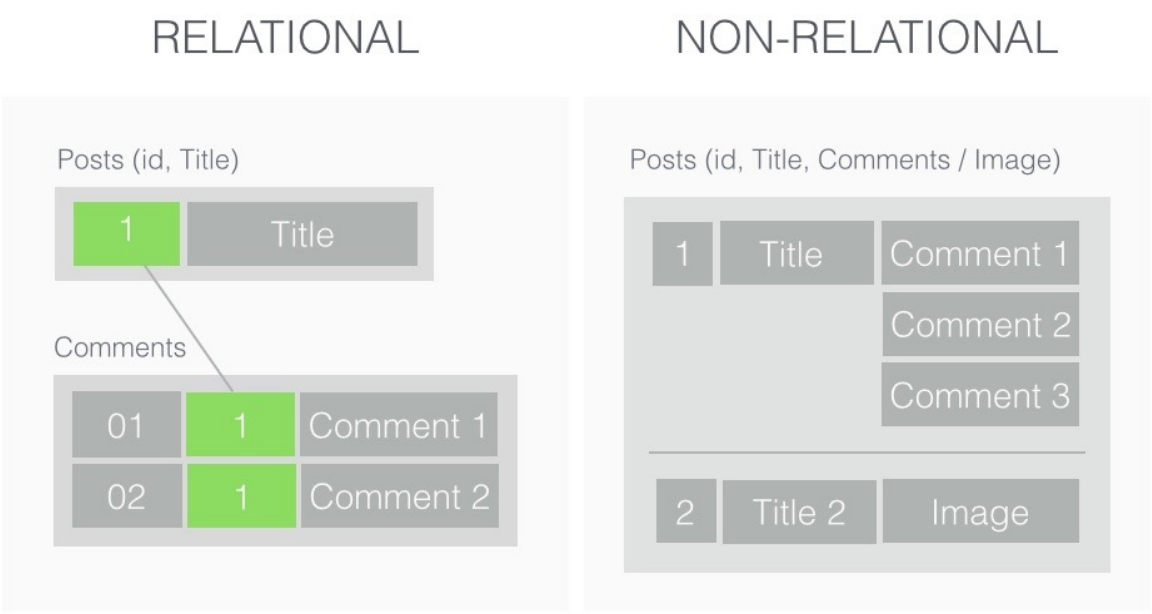

Figura 4.1: Esempio differenze fra schemi SQL e Documenti.

Fonte: www.upwork.com

Individuazione di casi problematici

Di seguito è mostrato un esempio di tipo dalla difficile mappatura in un database SQL:

```
type Person implements Account {
    id: ID!
    name: String!
    address: Address
    dog: Dog
    amount: Float
    dogSittersId: [ID!]
    dogSitters: [DogSitter!] @mongoInnerRef
    beagleId: ID!
    beagle: Beagle @mongoInnerRef
}
```

I campi seguiti dalla direttiva *@mongoInnerRef* non sono mappati sul database, ma verranno gestiti come associazioni con il meccanismo di associazioni fra DAO.

I problemi di tipi come questo sono:

- Il tipo estende un'interfaccia
- Al suo interno il tipo contiene un campo di un altro tipo: *Address* (questi tipi non rappresentano tabelle sulle quali effettuare richieste direttamente, ma sono tipi innestati nei tipi delle tabelle)
- Al suo interno il tipo contiene un campo di tipo interfaccia: *Dog*
- Al suo interno il tipo contiene degli array

Database di tipo SQL non hanno soluzioni uniformi fra le varie implementazioni (o non ne hanno per niente) per gestire oggetti innestati, array e gerarchie.

Anche l'interazione con i DAO che gestiscono tipi complessi come quello appena segnalato presenta problematiche: è infatti previsto, almeno nella versione mongoose, che si possa interagire con i DAO delle interfacce come se si stessero interrogando anche i DAO delle implementazioni. Per esempio, dati i tipi

```
interface User @mongoEntity {
  id: ID! @id
}
type CustomerUser implements User {
  id: ID!
  profile: ProfessionalProfile
}
```

mappati in tipi TypeScript nel seguente modo

```
export type User = {
  id: Scalars['ID'];
};
export type CustomerUser = User & {
  __typename?: 'CustomerUser';
  id: Scalars['ID'];
  profile?: Maybe<ProfessionalProfile>;
};
```

dovrebbe essere possibile effettuare la seguente operazione

```
let customerUser : CustomerUser =
  { id: user.id, __typename: "CustomerUser",
    firstName: "FirstName 1", organizationId: "organization_id",
    live: true };
await DAO.user.insert(customerUser);
```

ed aspettarsi, interrogando successivamente il DAO di user, di ricevere in risposta il record di tipo CustomerUser con tutti i suoi attributi, anche quelli non relativi alla tabella degli Users.

Risoluzione delle problematiche - alto livello

In un primo momento era stato ipotizzato di gestire i casi sopra descritti con delle associazioni, sfruttando il meccanismo di riferimenti già presente nel progetto dell'azienda. Questo approccio avrebbe però portato a dover gestire una serie di complicazioni:

- Generazione manuale delle tabelle di giunzione per gestire le relazioni fra tipi
- Necessità di effettuare modifiche all'interfaccia delle associazioni, quindi dover modificare l'interfaccia del DAO comune
- Dipendenza di associazioni l'una dall'altra: è per esempio possibile che un'associazione interna dipenda da un campo contenuto in un oggetto innestato; essendo, nel caso SQL, l'oggetto embedded rappresentato da un'altra tabella associata, sarebbe stato necessario definire un meccanismo per rilevare e gestire le dipendenze fra le due associazioni.

Sono stati invece evitati questi problemi usando le associazioni di Sequelize.

(nota: verrà usato il termine *riferimenti* per riferirsi alle associazioni indicate su schema GraphQL e risolte tramite il meccanismo interno al DAO, mentre *associazioni* per intendere associazioni Sequelize) Non usando il meccanismo di gestione dei riferimenti si evita di dover effettuare cambiamenti all'interfaccia comune e non bisogna gestire dipendenze fra essi, in quanto la loro risoluzione verrà eseguita solo dopo che il DAO Sequelize avrà raccolto tutti i dati della tabella e di quelle associate. Oltre a questo, Sequelize gestisce in autonomia (o dietro configurazione esplicita) la creazione di foreign keys per collegare le tabelle fra di loro, non rendendo necessario aggiungere questi campi, utili solo alla versione SQL, nello schema GraphQL.

I campi di tipi embedded verranno mappati in tabelle secondarie, con riferimenti alle tabelle alle quali appartengono. Gli array vengono gestiti tramite l'associazione Sequelize di tipo HasMany: nel caso siano array di dati scalari, verrà creata una tabella apposita per contenere questi valori.

Per la gestione delle interfacce è stato scelto di tenere tutti i campi comuni in una tabella relativa all'interfaccia, mentre in quelle delle implementazioni saranno salvati i dati relativi ad esse ed un riferimento al record associato sulla/e tabella/e dell'interfaccia/e. Viene aggiunto, dove non indicato nello schema GraphQL, un campo id fittizio. Per gestire le tabelle aggiuntive verranno inizializzati dei DAO dedicati, tenendoli però nascosti all'utilizzatore dell'applicazione.

Il tipo precedentemente mostrato e quelli associati verranno quindi mappati con la seguente struttura di tabelle

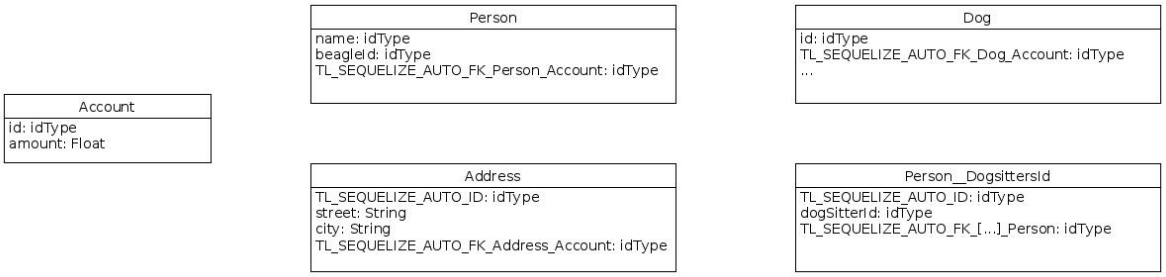

Figura 4.2: Schema delle tabelle per l'entità *Person*

I campi aggiuntivi sono quelli con la dicitura *TL_SEQUELIZE_AUTO*.Person_dogSittersId è una tabella aggiunta, che serve per contenere i valori dell'array *dogSittersId*.

Per gestire le associazioni fra DAO è stata definita la seguente interfaccia:

```
interface SequelizeAssociation {
  field : string,
  daoModelName : string,
  kind : SequelizeAssociationKind
  alias: string,
  extractField? : string,
}
```

Ogni DAO conserva una mappa che, al nome di un campo, associa una tupla composta dal DAO associato a quel campo e dalla SequelizeAssociation corrispondente. Il significato dei campi è il seguente:

- field: nome del campo associato ad un altro DAO
- daoModelName: nome del DAO associato
- kind: indica se il DAO associato rappresenta un tipo *embedded*, *interfaccia* o *implementazione*
- alias: alias dell'associazione SQL fra i modelli dei DAO
- extractField: opzionale. Se utilizzato in un'associazione di tipo embedded, indica che il/i model associati dovranno essere mappati al campo indicato. Nell'applicazione, verrà usato per gestire gli array di primitive.

Il principio di funzionamento generale dell'applicazione è quindi quello di rilevare quando un campo presente nei parametri della query è relativo ad un DAO associato e, di conseguenza, delegare la risoluzione di quel campo al DAO.

Per impedire modifiche parziali, le operazioni di modifica (inserimento, aggiornamento, rimozione) saranno effettuate all'interno di *transazioni* SQL. Se per esempio si verificasse un problema nell'inserimento di una sottoclasse, la parte di record contenuta nella tabella della superclasse non dovrebbe venire salvata.

Risoluzione delle problematiche - implementazione

Le funzioni principali del DAO astratto da implementare sono:

- Insert - Inserimento.
- Find - Ricerca
- Update - Aggiornamento
- Delete - Eliminazione

Tutte le altre operazioni (come *count* e *replace*) sono derivabili da queste.

Per gestire le associazioni, molti dei metodi scritti hanno una struttura ricorsiva del tipo:

```
function recursiveFunction(query, ...) {
  for(let field in query) {
    const a = this.sequelizeAssociations.get(field)
    if (a) {
      if(a is of kind EMBEDDED) {
        a.getDao().recursiveFunction(query[field])
      } else {
        a.getDao().recursiveFunction({field : query[field]})
      }
    }
    ...
  }
}
```

Se un'associazione viene rilevata su un campo del parametro query, la funzione verrà richiamata sul DAO associato a quel campo. Bisogna però fare attenzione al tipo di associazione: se è di tipo embedded, il DAO associato si occuperà di gestire direttamente il dato passato. Altrimenti, vuol dire che il DAO associato è una interfaccia o implementazione e potrebbe dover delegare nuovamente il campo ad un DAO associato in modo embedded.

Inserimento

Per l'inserimento è definita una sola funzione, che accetta il record da inserire nel database ed eventuali opzioni di scrittura. Sequelize permette di effettuare l'operazione di inserimento passando direttamente un oggetto composto, a patto di includere le associazioni alle tabelle relative nel parametro "options" di input.

Esempio di inserimento di un oggetto di tipo Product, uno User associato e indirizzi associati allo user.

```
return Product.create({
    title: 'Chair',
    user: {
        firstName: 'Mick',
        addresses: [{
            city: 'Austin',
        },{
            city: 'New York',
        }]
    }
    }, {
    include: [{
        association: Product.User,
        include: [ User.Addresses ]
    }]
});
```

L'inserimento viene quindi eseguito in due fasi: Nella prima, il record passato in input viene processato e tutti i suoi campi associati vengono raggruppati dietro all'*alias* delle associazioni. Nel caso dell'esempio precedente, in un oggetto di tipo

```
{
  title: 'Chair',
  firstName: 'Mick',
  addresses: [{city: 'Austin}, ...]
}
```

passato come input al DAO "product", vengono rilevate le associazioni ai campi "firstName" e "addresses" e raggruppate dietro l'alias ("user") dell'associazione, come mostrato nell'esempio sopra. Ricorsivamente, nel DAO "user", viene gestito il campo associato "addresses".

Nella seconda fase viene invece costruito l'oggetto delle opzioni che deve ricorsivamente indicare tutte le associazione incluse nella query.

Ricerca

Interfaccia della funzione Sequelize

Sequelize permette di effettuare un'operazione di ricerca includendo subito uno o più modelli associati, tramite il meccanismo di Eager Loading.

Esempio di semplice query di ricerca con un modello associato, nella quale viene indicato anche l'alias dell'associazione:

```
const users = await User.findAll({
  include: { model: Tool, as: 'Instruments' }
});
```

È possibile, in alternativa al passare la coppia di attributi "model"/"as", definire un campo "association", inizializzato con il nome dell'associazione.

```
User.findAll({ include: { association: 'Instruments' } });
```

È anche supportata l'inclusione di più modelli alla volta, nonché di modelli associati a quelli inclusi, in modo innestato.

```
User.findAll({ include: [
                          { association: 'Instruments' },
                          { association: 'Tools',
                            include : [
                              association : 'Designers'
                            ]
                          }
                        ]});
```

Tramite l'opzione *where* è invece possibile definire un filtro per la ricerca.

```
User.findAll({
  where : {firstName : 'Paul'},
  include: {
    association: 'Instruments',
    where: {
      size: {
        [Op.ne]: 'small'
      }
    }
  }
});
```

Definire così le opzioni di ricerca scollegherebbe i filtri dei campi di una tabella da quelli dei campi di una tabella che rappresenterebbe un oggetto innestato in quella precedente. Ad esempio, supponendo che l'associazione "Instruments" rappresenti un oggetto embedded in User, il filtro di ricerca

```
userFilter : UserFilter = {
  $or : {
    firstName : 'Paul',
    "Instruments.size" : {
      $ne : 'small'
    }
}}
```

Non sarebbe traducibile, non potendo mettere in "OR" i due filtri, secondo la modalità precedente presentata.

Fortunatamente, Sequelize permette di elevare allo stesso livello tutti i filtri, risolvendo così il problema.

```
User.findAll({
  where : {
    [Op.or] : {
      "$firstName$" : 'Paul',
      "$Instruments.size$" : {
        [Op.ne] : 'small'
      }
    }
  },
  include: {
    model: Tool,
    as: 'Instruments'
  }
});
```

È possibile indicare, per ogni modello incluso, una lista di attributi da estrarre. Grazie a questo è possibile implementare la proiezione di attributi.

L'ordinamento, similmente ai filtri, è specificabile sia modello per modello che al livello radice della query. Quest'ultima opzione è stata scelta. Ad esempio, l'ordinamento

```
userSort : UserSort = [
  {firstName : SortDirection.ASC},
  {"Instruments.size" : SortDirection.DESC}
]
```

può essere tradotto in una query Sequelize nel seguente modo:

```
User.findAll(
  {
    order: [
      ['firstName', 'ASC'],
      ['Instruments', 'size', 'DESC']
    ]
    include: {
      model: Tool,
      as: 'Instruments'
    }
  });
```

Gestione delle inclusioni

Sequelize permetterebbe di effettuare automaticamente un join fra il Model iniziale, tutti quelli associati e potenzialmente anche quelli associati a quelli associati tramite l'opzione, da inserire nella query, *attributes: {all : true, nested : true}*.

Questo semplificherebbe di molto il problema della gestione della query, ma ogni interrogazione comporterebbe un numero elevato di operazioni di join. Per ridurre il numero di join l'implementazione della ricerca cercherà di includere solo le tabelle strettamente necessarie.

A tal proposito, vengono analizzati filtri, proiezioni e ordinamenti ricevuti in input per identificare i modelli da includere nella ricerca. Per fare ciò, i parametri in input vengono tradotti, dove serve, sia in un formato comprensibile per Sequelize, sia in un oggetto a più livelli per facilitare l'individuazione di campi associati.

Per esempio, il filtro "userFilter" precedentemente mostrato viene convertito in

```
{
  $or : {
    firstName : 'Paul',
    Instruments : {
      size : {
        $ne : 'small'
      }
    }
  }
}
```

così da poter, rilevare, percorrendo ricorsivamente la struttura, l'associazione con il model Tool sul campo Instruments.

Aggiornamento

Sono rese disponibili tre funzioni di aggiornamento:

- update: accetta un oggetto di tipo ModelType, lo cerca di individuare nel database ed aggiornarlo
- updateOne: accetta un filtro di ModelType e aggiorna il primo record individuato in base ad esso
- updateMany: accetta un filtro di ModelType e aggiorna tutti i record individuati in base ad esso

Tutte e tre le funzioni ricevono anche in input un oggetto per definire gli aggiornamenti da scrivere.

L'operazione di aggiornamento con modelli associati non è correntemente supportata da Sequelize, quindi è stato implementato il seguente procedimento:

1. Viene eseguita un'operazione di ricerca per ottenere tutti i modelli da modificare in base al filtro passato alla funzione
2. Per ogni campo dell'oggetto che indica i cambiamenti, si controlla se il nome di questo campo corrisponde a quello di un'associazione
3. In caso positivo, per ogni modello da aggiornare, bisogna verificare se il campo di quel modello è un array
    - Nel caso lo sia, l'array va sostituito completamente con quello passato in ingresso. Vengono quindi cancellati tutti i modelli associati con il modello preso in esame e successivamente inseriti e associati nuovi modelli sulla base dei cambiamenti passati.
    - Altrimenti, se il campo non è vuoto, la funzione di aggiornamento viene richiamata ricorsivamente passando come parametri i cambiamenti sul campo corrente e il modello da aggiornare (sempre facendo caso al tipo di associazione rilevata)

4. Alla fine, vengono mappati tutti i modelli da aggiornare ai loro id, per poter eseguire un'unica operazione di aggiornamento in blocco sulla tabella sfruttando come filtro la lista di id, piuttosto che chiamare la funzione di aggiornamento per ogni modello (effettuando quindi n operazioni di update).

Eliminazione

Le funzioni di eliminazione da implementare sono due:

- deleteOne: funzione che riceve in input un oggetto e cerca di eliminare il record equivalente nel database
- deeteMany: funzione che riceve in input un filtro e cerca di eliminare tutti i record del database che lo soddisfano

L'eliminazione di un modello e di tutte le sue associazioni di tipo hasOne/hasMany è gestita automaticamente da Sequelize impostando nell'associazione l'opzione sql ON-DELETE CASCADE. Questo però non copre due casi: quello in cui si chiami la funzione di eliminazione sul DAO di una implementazione e quello in cui lo si chiami su quello di una interfaccia la quale implementazione implementa due o più interfacce.

Nel primo caso verrebbe eliminato solo il record dell'implementazione mentre quelli relativi alle interfacce no, poiché associati con associazioni di tipo "belongsTo", mentre nel secondo il record dell'interfaccia e dell'implementazione verrebbero eliminati, ma quelli delle altre interfacce no, non essendo associati alla prima.

Visto che solo i record delle implementazioni conoscono tutte le interfacce ad essi collegati, sono i DAO di questi a gestire le operazioni di eliminazione: se la funzione viene chiamata sul DAO di un'interfaccia, questo la rigira a tutti i DAO rilevanti di implementazioni. I DAO delle implementazioni poi si occuperanno di distruggere i record corretti sui modelli delle interfacce.

L'implementazione dei modelli Sequelize della funzione di delete accetta fra i parametri un filtro per indicare quali record andranno eliminati, ma non permette però di includere nel filtro campi di altre tabelle (come nel caso della ricerca). Nel caso dell'operazione deleteMany quindi, sarà prima necessario recuperare tutti i models da eliminare con una ricerca per poi mapparli sui loro id, così da rendere il filtro per l'operazione di delete corretto, lo stesso procedimento viene effettuato anche sulle interfacce.

Capitolo 5

Sviluppo del generatore

Il sistema per la generazione dei modelli è stato suddiviso, seguendo la struttura già adottata dall'azienda, nei seguenti moduli:

- Un plugin per GraphQL-code-generator per tenere in comunicazione gli altri moduli
- Un file per gestire il trattamento dei nodi del modello GraphQL (implementando il pattern visitor)
- Un generatore per verificare la validità dei nodi e invocare metodi di generazione
- Un generatore astratto per contenere metodi comuni e le sue implementazioni:
  - Un generatore per generare gli schemi Sequelize
  - Un generatore per generare DAO e tipi di supporto

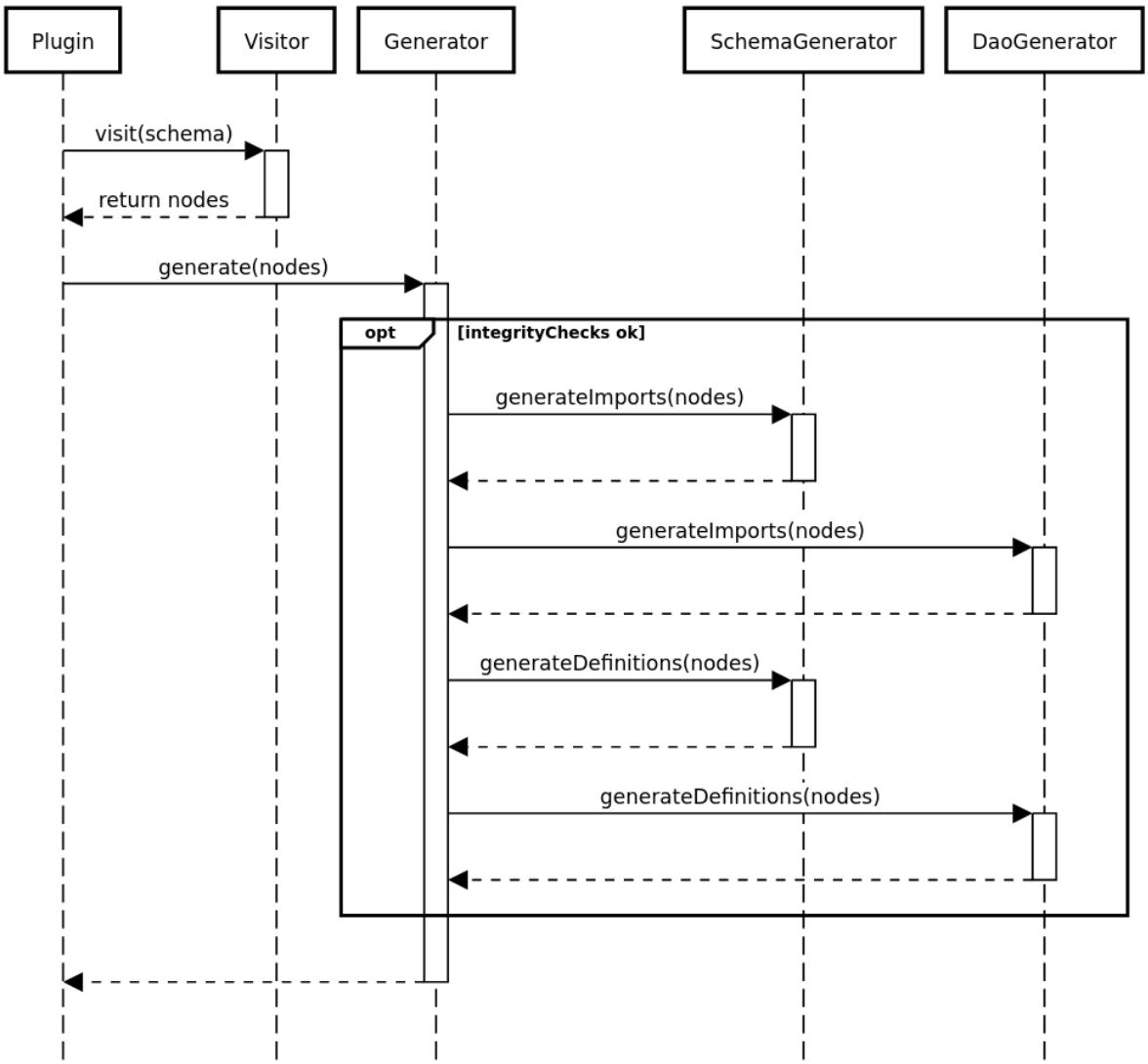

Figura 5.1: Flusso di comunicazione fra i vari moduli del generatore

Modulo visitor Viene definito un oggetto che estende la classe *BaseVisitor* di GraphQL-codegen ed implementa metodi *visit* per gestire definizioni di tipi ed interfacce. Nodi interfaccia e tipo vengono trattati in modo simile e il loro trattamento genera uno o più oggetti (nel caso sia necessario creare tabelle di supporto per quella principale) di tipo *TsSequelizeGeneratorNode*.

```
export enum NodeTypes {
  INTERFACE,
  TYPE,
  EMBEDDED
}
export type TsSequelizeGeneratorNode = {
  type?: NodeTypes
  name: string
  isAutoGenerated: boolean
  collection: string
  interfaces: string[]
  fields: TsSequelizeGeneratorField[]
  ...
}
```

Campi principali del tipo TsSequelizeGeneratorNode:

- type: è assente se il nodo è autogenerato (vedere in seguito), altrimenti prende uno dei valori dell'enum NodeTypes.
- name: nome del nodo
- isAutoGenerated: identifica se il nodo è stato aggiunto pur non essendo presente nello schema GraphQL. In particolare, identifica i nodi creati per gestire array di dati primitivi.
- collection: nome della tabella nel database
- interfaces: nome delle interfacce del nodo
- fields: collezione contenente i campi del nodo trattato

Modulo generatore Il modulo di generazione, invocato dal plugin dopo aver raccolto tutti i nodi trattati, analizza ognuno di questi e ne controlla la correttezza. In particolare, viene verificato che:

- Ogni nodo abbia un campo id o almeno una interfaccia con un campo tale.
- I campi sui quali i riferimenti (segnalati tramite direttiva @sqlizeForeignRef o @sqlizeInnerRef) si basano siano esistenti.
- I campi condivisi fra implementazioni ed interfacce siano di tipo compatibile.

Dentro la classe del generatore è contenuta una lista di implementazioni di generatore del file finale, tutti implementanti una classe astratta. Dopo aver controllato tutti i punti precedenti, il generatore principale invoca i metodi di generazione di imports, definizioni ed exports dei generatori della lista.

Generatore degli schemi db Il generatore degli schemi di Sequelize analizza tutti i nodi, generando per ciascuno un model Sequelize. Gli attributi del model generato corrispondono ai campi del nodo che:

- Non hanno direttive di tipo "reference" nello schema
- Non appartengono ad interfacce
- Non sono etichettati come liste
- Non sono etichettati come di tipo embedded

Generatore dei DAO Per ogni nodo generato dal visitor, il generatore dei DAO verifica in primo luogo se l'attributo "isAutoGenerated" è impostato. Nel caso lo sia, significa che il nodo non è parte dello schema GraphQL e che quindi il plugin di generazione dei tipi TypeScript, lanciato prima del plugin personalizzato descritto in queste sezioni, non ha generato un tipo relativo ad esso. Il generatore dei DAO deve quindi definire tipi per gestire i nodi autogenerati poiché gli oggetti di tipo SequelizeAbstractDAO richiedono fra i parametri generici anche il tipo del DTO.

Il generatore dei DAO si occupa anche, per ogni nodo, di generare filtri, proiezioni e ordinamenti sulla base dei campi del nodo.

Infine, per ogni nodo, genera l'oggetto di tipo DAO, inizializzandolo con i corretti riferimenti e sequelizeAssociations.

Dopo aver generato tutti i DAO, il generatore definisce anche il DaoContext, oggetto che inizializza i DAO e ne contiene un riferimento. È il DaoContext che, ricevuta in input un'istanza di oggetto Sequelize, inizializza tutti i modelli definiti dal generatore degli schemi e le loro associazioni Sequelize.

Capitolo 6

Valutazione dei risultati

Testing

Una volta superate le fasi di studio iniziale e prodotto un primo prototipo, parte del processo di sviluppo dei tools e del DAO astratto è stato dedicata al testing.

A tal scopo, è stato definito manualmente un ambiente di prova che simulasse un possibile output finale finale del generatore. In particolare, è stata prestata attenzione alla creazione di tipi che potessero essere usati per testare alcuni punti critici:

- tipi con tipi embedded fra i loro campi;
- tipi con tipi embedded fra i loro campi, a loro volta composti da altri tipi innestati;
- tipi con array, sia di primitive che di tipi, fra i loro campi;
- tipi che estendono una interfaccia;
- tipi che estendono più interfacce;
- interfacce estese da più tipi;

Per questo ambiente sono stati poi scritti una serie di *unit tests*, utilizzando la libreria di testing JavaScript *Jest*, già impiegata nelle prove del prototipo aziendale vista la sua semplicità di utilizzo ed integrazione.

Questi test sono stati scritti per verificare il comportamento delle operazioni del DAO sia in caso di manipolazioni di dati di tipo semplice, sia in caso di tipi più complessi, come quelli elencati precedentemente. Vengono di seguito illustrati i casi più interessanti, anche per dimostrare il funzionamento del prodotto della tesi.

Gli esempi seguenti interagiranno con i DAO *user*, *user2* e *subUser*: il primo definisce una interfaccia con gli attributi *firstName, a, b*, il secondo ha l'attributo *firstName2* mentre subUser è un tipo che implementa le due interfacce e aggiunge l'attributo *subAttr*.

```
type User = {
    id?: Scalars['ID'];
    a?: Maybe<A>;
    b?: Maybe<Array<Scalars["String"]>>;
    firstName?: Maybe<Scalars['String']>;
};

type User2 = {
  id?: Scalars['ID'];
  firstName2?: Maybe<Scalars['String']>;
};

type SubUser = {
  id?: Scalars['ID'];
  a?: Maybe<A>;
  b?: Maybe<Array<Scalars["String"]>>;
  firstName?: Maybe<Scalars['String']>;
  firstName2?: Maybe<Scalars['String']>;
  __typename? : "SubUser";
  subAttrib?: Maybe<String>
};
```

Inserimento

Entità innestate e array

```
const record = {
                firstName: "FirstName",
                a : {
                      a : "A",
                      a2 : "A2",
                      d: {
                          d : "D",
                          e : ["E"]
                         }
                    },
                b: ["B1", "B2", "B3"]
               })
const user = await dao.user.insert(record);
expect(user.a).toBeDefined();
expect(user.a?.a).toBe("A");
expect(user.a?.d?.d).toBe("D");
expect(user.a?.d?.e).toHaveLength(1);
expect(user.b).toBeDefined();
expect(user.b).toHaveLength(3);
```

Al metodo *insert* di un DAO viene passato un oggetto composto da:

- campi di tipo scalare;
- campi di tipo oggetto;
- campi di tipo array.

Fra i campi di tipo oggetto ce ne sono anche alcuni a loro volta composti da oggetti. Si verifica, dopo l'inserimento, che tutti i campi siano esistenti e corretti.

Sottoclassi

```
const record = {
                  firstName: "FirstName",
                  subAttrib: "subAttr"
                }
await dao.subUser.insert(record);
const user = await dao.user.findOne({});
const subUser = await dao.subUser.findOne({});
expect(subUser!.firstName).toBe(user!.firstName);
expect(subUser!.id).toBe(user!.id);
expect(subUser!.subAttrib).toBe("subAttr");
```

Si chiama il metodo di inserimento di un DAO di tipo sottoclasse (ovvero un tipo che implementa un'interfaccia). Si verifica successivamente che sia possibile recuperare lo stesso record inserito sia interrogando il DAO dell'interfaccia che quello dell'implementazione. La prova di uguaglianza fra il record ottenuto dal DAO dell'interfaccia e quello dell'implementazione viene fatta su una serie di attributi, fra cui l'ID.

Classe con più interfacce

```
let su : SubUser = {
                    firstName : "subName",
                    firstName2 : "fname2",
                    subAttrib : "subAttr",
                    b: ["b1", "b2"]
                   }
su = await dao.subUser.insert(su);
const u = await dao.user.findOne({});
const u2 = await dao.user2.findOne({});
expect(su.id).toBe(u?.id);
expect(su.id).toBe(u2?.id);
```

Viene inserito un record, tramite il DAO di una sottoclasse, che implementa più di una interfaccia. Successivamente si controlla che interrogare il DAO utilizzato o quello delle sue interfacce porti a recuperare lo stesso record.

Aggiornamento

Aggiornamento di entità innestate

```
\\[...] inserimento records
dao.user.updateMany({
                      "a.d.d" : "D",      //Filtro
                      "a.d.e" : null
                     }, {
                      "a.d.d" : "Dc",     //Aggiornamenti
                      "a.a" : "A11",
                      "a.d.e" : "E2",
                      "b": ["a", "b"]
                     });

const numUpdated = await dao.user
                             .count({
                               "a.d.d" : "Dc",
                               "a.a" : "A11",
                               "a.d.e" : "E2",
                               b : ["a", "b"]
                             })
expect(numUpdated).toBe(2)
```

Vengono inseriti dei records con oggetti innestati e array e poi, sempre sullo lo stesso DAO dell'inserimento, viene chiamata la funzione di aggiornamento, indicando che si vogliono aggiornare solo alcuni dei record aggiunti utilizzando il parametro di tipo filtro. I cambiamenti indicati vanno a definire aggiornamenti sia su array che su oggetti, incluso l'aggiornamento di parametri di oggetti innestati dentro oggetti innestati. Successivamente si contano i record che sono stati correttamente aggiornati.

Aggiornamento di sottoclassi

```
let user = await dao.subUser
                    .insert({ firstName: "FirstName" });
user = (await dao.subUser.findOne({ id: user.id }));
await dao.subUser.update(user, {
                                 subAttrib : "subAttrib",
                                 firstName: "FirstName1"
                                });
const user2 = await dao.subUser.findOne({ id: user.id });
expect(user2!.firstName).toBe("FirstName1");
expect(user2!.subAttrib).toBe("subAttrib");
```

Viene inserito un record di tipo "sottoclasse" e poi lo si aggiorna. Fra i campi aggiornati ce ne sono anche di appartenenti all'interfaccia implementata dal record.

Sostituzione

Sostituzione di superclasse con sottoclassi

```
let user = await dao.user
                    .insert({ firstName: "FirstName" });
let subUser : SubUser = {
                          id: user.id,
                          firstName: "FirstName 1",
                          subAttrib: "subAttrib"
                        };
await dao.user.replace(user, subUser);
user = await dao.user.findOne({ id: user.id });
expect(user).toBeDefined();
expect(user?.firstName).toBe("FirstName 1");

\\[...] definizione type guard isSubUser

if(isSubUser(user)){
    expect(user.subAttrib).toBe("subAttrib");
} else {
    fail("Replaced user is not of expected subclass.")
}
```

Viene inserito un record tramite il DAO della superclasse, poi sostituito con un oggetto di tipo "sottoclasse1". Dopodiché, tramite una ricerca per ID, si accede al nuovo record e ne si controlla il tipo, che deve quindi essere diventato "sottoclasse1". Si sostituisce nuovamente il nuovo record con uno di tipo "sottoclasse2" e si ripete la prova.

Eliminazione

Eliminazione di classe con più superclassi

```
const su : SubUser = {
                      firstName : "subName",
                      firstName2 : "fname2",
                      subAttrib : "subAttr"
                     }
\\[...] inserimento records
await dao.subUser.deleteMany({
                               firstName : "subName",
                               subAttrib : "subAttr"
                              })
expect(await dao.user2.count({})).toBe(0)
```

Vengono inseriti records di tipo "sottoclasse1" (che implementa due interfacce) e successivamente li si eliminano chiamando la funzione di eliminazione sul DAO di "sottoclasse1". Fra i parametri del filtro di eliminazione sono inseriti anche attributi appartenenti ad una delle due interfacce. Si verifica poi, tramite una ricerca sul dao della seconda interfaccia, che siano stati correttamente eliminati tutti i records.

Limiti dell'implementazione proposta

Le associazioni salvate nella lista "sequelizeAssociations" di un Dao sono identificate solo tramite il nome di un campo, pertanto non è possibile gestire il caso in cui un DAO abbia più relazioni basate su campi con lo stesso nome. È questo per esempio il caso di una interfaccia implementata da due implementazioni (o viceversa) che condividono un campo dal nome uguale ma non presente nell'interfaccia: il DAO dell'interfaccia potrà avere una sola sequelizeAssociation basata sul campo condiviso dalle due interfacce.

Viene mostrato un esempio di inserimento nel quale questo problema si manifesta:

```
interface Account {
  id: ID! @id
  amount: Float
  address: Address
  dog: Dog @sqlizeForeignRef(refFrom: "accountId")
}
type Person implements Account {
  id: ID!
  amount: Float
  address: Address
  dog: Dog
  name: String!
  surname: String!
}

type Business implements Account {
  id: ID!
  amount: Float
  address: Address
  dog: Dog
  name: String!
  vatNumber: String!
}
```

I due tipi che implementano l'interfaccia "Account" hanno entrambi un campo con lo stesso nome ma non appartenente all'interfaccia: "name". Il DAO del tipo "Person" verrà generato e inizializzato per primo, dunque "Account" avrà una sequelizeAssociation basata sul campo "name" e collegata con il DAO "Account".

Se successivamente si provasse ad inserire un record di tipo "Business" chiamando il metodo di inserimento sul DAO di "Account", quest'ultimo cercherebbe di inserire due record: uno nella tabella "Business" contenente l'attributo "vatNumber" e uno nella tabella "Person" composto dall'attributo "name".

Questa limitazione è dovuta all'utilizzo, nel progetto originale, di tipi Typescript piuttosto che di classi: non potendo riconoscere il tipo dell'oggetto passato in input se non analizzando i suoi campi, collisioni di questo tipo sono difficilmente gestibili.

I tipi TypeScript generati dal plugin ufficiale di GraphQL-codegen hanno un campo nominato "__typeName", contente il nome del tipo concreto (quindi non delle interfacce), con il quale si potrebbe risolvere il problema. Nella configurazione usata dall'azienda però questo campo è reso opzionale, pertanto un utente potrebbe inserire un record senza specificare il tipo dell'oggetto passato in input.

Capitolo 7

Conclusioni

Risultati

In questa tesi sono stati raggiunti gli obiettivi base del lavoro: sono stati implementati sia il DAO astratto in grado di comunicare, tramite libreria Sequelize, con lo strato del DB, che il plugin per la generazione di DAO, compresi filtri, proiezioni, ordinamenti e Modelli, implementanti il DAO astratto.

È stata data considerazione alle prestazioni, cercando di eseguire dove possibile meno operazioni di JOIN per le operazioni di lettura e, per quelle di scrittura, più operazioni di scrittura in blocco.

La comunicazione con l'azienda è stata chiara e utile per dissipare eventuali dubbi sulla struttura del progetto o su certi aspetti di MongoDB, database del quale il sottoscritto non ha esperienza.

I limiti della soluzione già evidenziati non sono stati ritenuti particolarmente preoccupanti, in quanto sono state rilevate problematiche in casi simili anche per la versione MongoDB.

Lavori futuri

Oltre ad un supporto continuato (tutto il progetto, comprese le interfacce del DAO, è ancora in fase prototipale), una funzionalità immediatamente aggiungibile sarebbe quella di poter far scegliere, in fase di generazione, come trattare le gerarchie di interfacce e implementazioni. Correntemente è possibile gestire le gerarchie solo nel modo descritto nei capitoli precedenti, ma l'azienda sarebbe interessata a poter gestire le gerarchie anche negli altri due modi rimanenti:

- Accorpando tutte le implementazioni nella tabella dell'interfaccia
- Generando solo le tabelle delle implementazioni, ognuna contenente anche i campi delle interfacce

Bibliografia

[1] GraphQL official site,
https://graphql.org/learn/

[2] GraphQL code generator official site,
https://graphql-code-generator.com/docs/getting-started/index

[3] Sequelize documentation,
https://sequelize.org/master/

[4] Bran Selic,
Model-Driven Development: Its Essence and Opportunities