

Sviluppo di un generatore di codice per librerie ORM SQL basato su schemi GraphQL

Candidato
Ivan Perazzini

Relatore
prof. Mirko Viroli

Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

19 marzo 2021

Outline

- 1 Introduzione
- 2 GraphQL
 - Storia
 - Caratteristiche
- 3 Progetto
 - Architettura
 - Obiettivi
 - Svolgimento
- 4 Risultati

Twinlogix

Twinlogix S.r.l, azienda di Santarcangelo di Romagna, ha seguito con interesse la maturazione dello standard GraphQL ed in questo momento sta ponendo fra gli obiettivi interni il passaggio completo da API Rest a GraphQL.

Un altro progetto legato a questo è quello di sfruttare le potenzialità del linguaggio GraphQL per costruire strumenti di generazione di codice basati su di esso. Questi strumenti, basandosi su uno schema GraphQL, dovranno generare definizioni di modelli per database e relativi DAO.

Dato che un prototipo di generatore (per database MongoDB) è stato già scritto dall'azienda, l'obiettivo di questa tesi sarà studiare ed implementarne una versione per database SQL.

GraphQL

È un linguaggio per API ideato per descrivere operazione di lettura e scrittura di dati definito da Facebook nel 2012 ed dal 2018 diventato standard gestito da *GraphQL Foundation*. La sua nascita è dovuta a delle necessità dell'azienda statunitense.

- i dati gestiti da Facebook come post, utenti, commenti e reazioni risultano meglio rappresentabili da database non relazionali;
- allo stesso tempo, il valore di questi dati risiede nelle relazioni fra di essi (adesempio, quali utenti sono amici di altri).

I database relazionali (in questo caso, documentali) non presentano nativamente meccanismi di relazione fra documenti diversi. Non esiste un linguaggio di querying unificato per database NoSQL.

Il linguaggio GraphQL permette di definire uno schema costituito da:

Oggetti Struttura delle entità gestite dall'applicazione, composte da campi tipizzati.

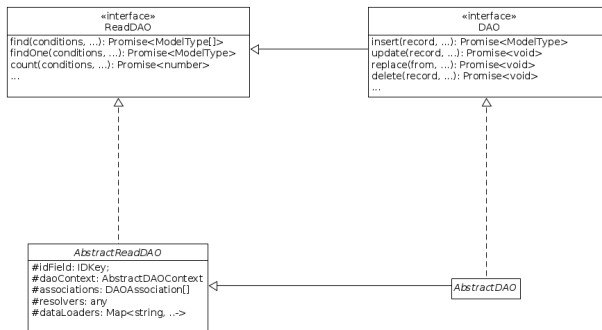
```
type Address {  
  street: String  
  city: String  
  country(continent: Continent): Country!  
}
```

Query e mutazioni Operazioni di lettura e scrittura esposte al client.

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

Per arricchire l'espressività dello schema è possibile usare *direttive*, parole chiave definite nello schema.

Architettura del progetto

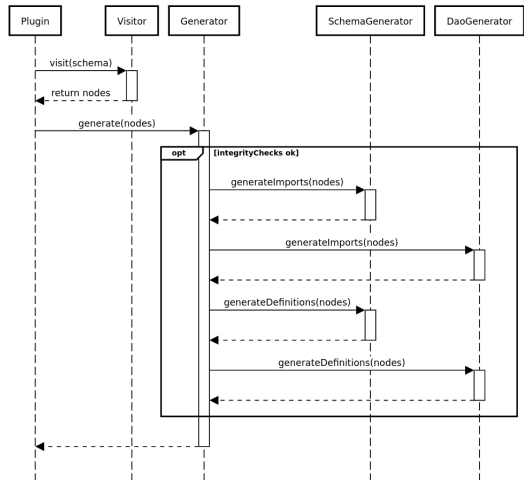


Per gestire l'interazione con le entità generate l'azienda ha impiegato il pattern *Data Access Object*, definendo un DAO generico.

Le implementazioni astratte di questo DAO si occupano di effettuare le operazioni su Database, interfacciandosi con driver e librerie apposite.

Per la generazione del codice è stato scritto un plugin per l'applicativo *GraphQL code generator* composto da:

- un modulo *visitor* per elaborare tipi ed interfacce dello schema;
- un modulo per controllare l'integrità dei tipi rielaborati;
- un generatore per gli schemi del database;
- un generatore per i DAO.



Esempi di funzionamento

Una volta generati i modelli delle tabelle, i DAO e i tipi di supporto al tipo di dato (filtri, proiezioni e ordinamenti), i DAO generati possono svolgere operazioni sulle tabelle del database.

```
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);
```

```
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"]  
});
```

Figura: Esempio di aggiornamento

Figura: Esempio di inserimento

Soluzione adottata - alto livello

```

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

```

Account
id: idType
amount: Float

Person
name: String
beagleId: idType
TL_SEQUENTIALIZE_AUTO_FK_Person_Account: idType

Address
TL_SEQUENTIALIZE_AUTO_ID: idType
street: String
city: String
TL_SEQUENTIALIZE_AUTO_FK_Address_Account: idType

Dog
id: idType
TL_SEQUENTIALIZE_AUTO_FK_Dog_Account: idType
...

Person_DogsittersId
TL_SEQUENTIALIZE_AUTO_ID: idType
dogSitterId: idType
TL_SEQUENTIALIZE_AUTO_FK_..._Person: idType

I campi di tipi embedded vengono mappati in tabelle secondarie, con riferimenti alle tabelle di appartenenza. Gli array vengono gestiti tramite associazioni "uno a molti": per array di scalari, vengono create tabelle apposite per contenerne i valori.

Per i tipi implementanti interfacce vengono tenuti tutti i campi comuni nella tabella dell'interfaccia.

Soluzione adottata - implementazione

Per la gestione della connessione e operazioni su db, è stata usata la libreria ORM Sequelize.

Le gerarchie e le composizioni sono modellate tramite le *associations* di Sequelize. Ogni DAO conserva al suo interno una mappa che associa nomi di campi ad associazioni.

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

Se durante una operazione vengono rilevati campi appartenenti ad una associazione, la gestione dell'operazione su quei campi è delegata ai DAO di competenza.

```
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]})  
      }  
    }  
    ...  
  }  
}
```

Figura: Struttura tipo dei metodi del DAO

Modelli testati

Il generatore è stato testato sullo stesso schema usato dall'azienda nel suo prototipo per MongoDB, mentre al DAO sono state fatte eseguire operazioni di scrittura e lettura su un modello composto da:

- 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;

Limitazioni

Le associazioni salvate in 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.

```
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!  
}  
  
interface Account {  
  id: ID! @id  
  amount: Float  
  address: Address  
  dog: Dog @sqlizeForeignRef(refFrom: "accountId")  
}
```

Grazie dell'attenzione.