ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA CAMPUS DI CESENA

DIPARTIMENTO DI INGEGNERIA DELL'ENERGIA ELETTRICA E DELL'INFORMAZIONE "GUGLIELMO MARCONI"

CORSO DI LAUREA IN INGEGNERIA BIOMEDICA

Studio e sperimentazione del linguaggio di query GraphQL

Tesi di Laurea in

CALCOLATORI ELETTRONICI

Relatore

Prof Luca Roffia

Correlatore

Elisa Riforgiato

Presentata da

Edoardo Lazzari

Anno accademico 2020-2021

Sommario

Abstract

GraphQL è un linguaggio di Query per lo sviluppo di API. Nello studio proposto di seguito verrà analizzato tale linguaggio nelle sue parti principali, per poi fare qualche esemplificazione sulla scrittura di *resolvers*, ovvero funzioni che fanno funzionare il *linguaggio Query* GraphQL. In particolare, verranno sviluppati *resolvers* basati su due database: **Mongodb** e **Neo4j**.

Indice delle figure

1 Introduzione

Al giorno d'oggi la gestione di dati, visualizzazione e modifica di essi, in particolar modo attraverso applicativi web, è tra le tematiche più studiate e utilizzate, in quanto sono di fondamentale importanza per la maggior parte dei campi di studio e lavoro. GraphQL è un moderno linguaggio di Query, creato per lo sviluppo di API (Application Programming Interface), che fornisce una sintassi molto flessibile per descrivere la richiesta e interazione tra i dati. GraphQL nasce nel 2012 dal gruppo di sviluppo di Facebook e passa successivamente, nel 2019, sotto il diritto d'autore di GraphQL Foundation, una fondazione neutrale fondata da varie aziende di informatica, che sostiene, e incoraggia la diffusione di GraphQL. Come progetto, GraphQL, viene ideata nel 2012, quando Facebook, stanca dei crash ripetuti, decide di reinventare il suo applicativo web; prima di ciò Facebook si basava su una architettura RESTfull e tabelle di dati FQL (una versione di SQL specifica di Facebook). Il team formato da Lee Byron, Nick Schrock e Dan Schafer ha quindi ripensato a come organizzare i dati, ma questa volta volgendo il proprio interesse verso gli utenti. La forza principale di GraphQL consiste, nel fatto che esso da semplicemente una struttura ai dati, definendo come essi sono descritti e come interagiscano fra loro, dicendo quindi come essi possano essere richiesti o modificati, lasciando aperta la possibilità di interagire con qualunque tipo di fonte di dati [1].

Lo scopo di questa tesi è una panoramica generale sul funzionamento dei resolvers, il codice in sé che fa operare le funzioni definite nelle API scritte in GraphQL, e in particolare un'applicazione di esempio in cui la fonte di dati sono due database: Mongodb e Neo4j.

2 Risorse utilizzate

Di seguito sono introdotte tutte le risorse che vengono poi utilizzate nello studio proposto.

2.1 GraphQL playground

GraphQL playground è un potente strumento per interagire con API GraphQL, è possibile utilizzarlo per fare richieste in linguaggio GraphQL e ricevere risposte. È stato sviluppato dal team Prisma, e tutte le informazioni possono esser trovate al link github: https://github.com/graphql/graphql-playground. Nello studio proposto le varie prove verranno fatte tutte sulla versione desktop, ma è utilizzabile anche in versione online, visualizzando la finestra di base si otterrà ciò che si può vedere in *figura 1*; nella parte sinistra si potranno scrivere le richieste, il cui risultato sarà visibile nella parte destra. Schema e Docs servono a visualizzare la struttura dei dati che il programmatore ha dato all'API in maniera da conoscere tutte le richieste permesse [2].

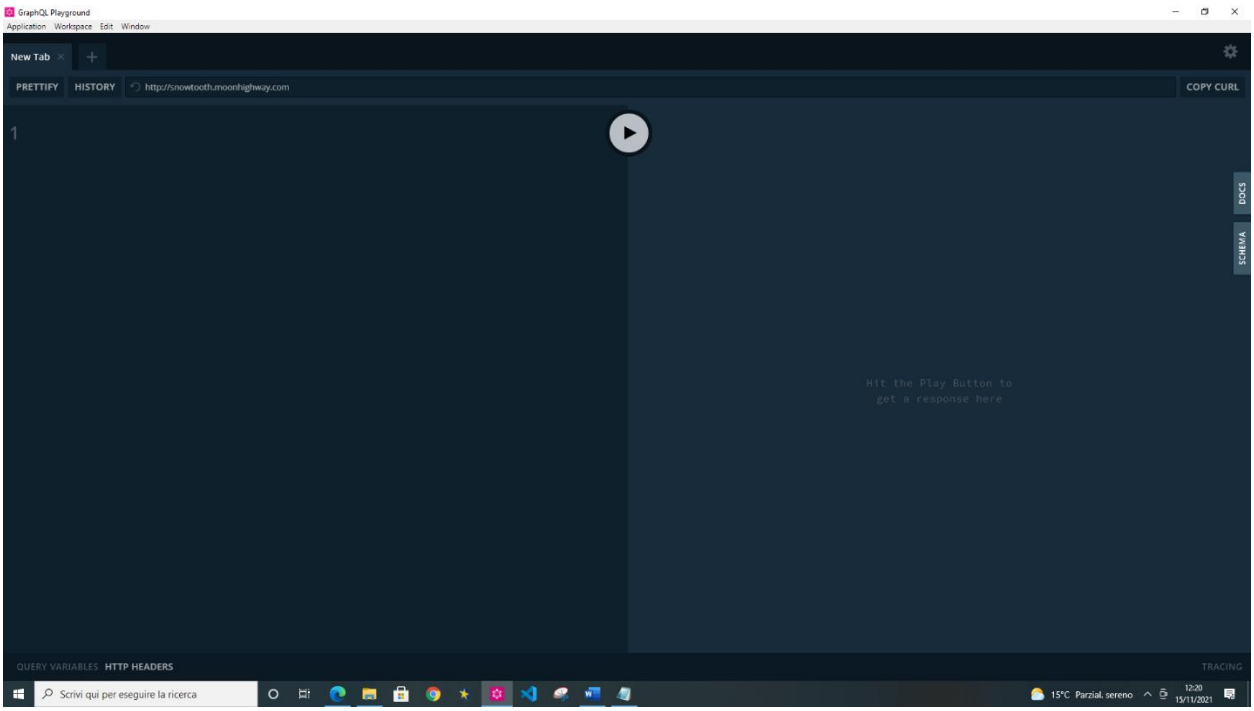

Figura 1 schermata GraphQL playground

2.2 GraphQL

GraphQL è al centro dello studio di questa tesi, esso corrisponde ad una serie di protocolli, che vengono messi tra il database, o fonte di dati generica, e l'applicativo Web, permettendo una facile comunicazione tra essi. La cosa molto interessante di GraphQL è che in un'unica API: utilizzando GraphQL, si possono prendere dati da più database anche diversi tra loro.

I principi di design di un API costruita con GraphQL sono 5 [1]:

- **Hierarchical**: i dati ricercati in una Query in GraphQL sono collegati l'uno all'altro, all'interno di una struttura gerarchica, e i dati in uscita sono visualizzati con la stessa struttura con la quale sono stati ricercati.
- **Product centric**: un'API definita in GraphQL, può essere scritta nella migliore maniera flessibile, in base alle necessità del servizio che si vuole offrire.
- **Strong typing**: ogni server GraphQL definisce un type system specifico all'applicazione. Le Query vengono eseguite nel contesto di type system. Data una Query, gli strumenti del server controllano la sua validità sintattica.
- **Client-specified queries**: le Query definite, sono scritte in maniera che, l'utente riceva solamente i dati richiesti e nulla di più, ovvero in risposta ad una Query vengono restituiti molti dati, attraverso GraphQL è possibile richiederne solo alcuni, a differenza delle architetture REST
- **Introspective**: il linguaggio Query di GraphQL può richiedere anche lo stesso type system del server GraphQL.

2.2.1 Schema

Utilizzando GraphQL, come metodo per costruire APIs, inizieremo a pensare a quest'ultime, non più come un insieme di punti REST, ma come un insieme di *types*. Per costruire un API, tutti questi *types*, vanno definiti formando uno schema che indica gli oggetti del server, le possibili modifiche e ricerche, insieme alla connessione tra gli oggetti; tutto questo lo si fa attraverso lo *schema* [3]. Inoltre, è importante sottolineare che, all'interno di un'API, qualunque tipo di database si utilizzi, lo schema non cambia, questo è uno dei motivi per cui un API può prendere dati anche da database differenti. Il linguaggio in cui si definisce lo schema GraphQL si chiama *SDL*[1], e anche questo, come il linguaggio Query, è sempre uguale per qualunque linguaggio di programmazione [3]. L'unita base di ogni schema GraphQL, è il *type*, esso, definisce un oggetto nel server, ognuno dei quali è caratterizzato da *fields*, essi rappresentano, appunto, i campi che descrivono un singolo type. Nella dichiarazione di un *type*, dopo il nome dei singoli fields, vanno usati i due punti e in seguito vengono dichiarati i tipi di dato essi contengono; INT, FLOAT, STRING, BOOLEAN ed ID sono le possibilità standard che vengono utilizzate come tipi di dati [3]. Se quest'ultimi sono tra parentesi quadre, vuol dire che quel *field* è un array formato da un insieme di dati di quel tipo, mentre il *punto esclamativo* è utilizzato per indicare se il field può essere anche null o no. Un esempio generale di dichiarazione di un *type* può essere il seguente descritto dalla *figura 2*.

```
type nome_type {
    nome_field1 : tipo_di_dato
    nome_field2 : tipo_di_dato_non_nullable!
    nome_field3 : [array_di_dati]
    }
```

Figura 2 definizione di un type

I tipi di dato precedentemente dichiarati, però non sono gli unici possibili che possono definire dei fields, ma possiamo ad esempio crearne alcuni noi. Un esempio sono gli *enum type*, ovvero una previa dichiarazione in cui creiamo un tipo di dato, appunto un *type*, che utilizzato come tipo di dato di un field restringe esso ad un limitato numero di valori. Per crearlo basta scrivere *enum* il nome del tipo di dato e tra parentesi graffe l'insieme di possibilità che vogliamo il dato abbia. Ciò si può fare come è esplicitato in *figura 3.*

[1] Schema Definition Language [1]

```
enum nome_enum_type{
    Primo_parametro
    Secondo_parametro
    Terzo_parametro
}
```

Figura 3 Definizione di un enum

Un altro esempio di tipo di dato con cui possiamo definire un field è, propriamente, un altro *type*, ovvero un altro oggetto del server. Questa potente modalità di programmazione è ciò che permette un collegamento tra i vari *types* di un API pensata con GraphQL e, anche in questo caso, vale la stessa regola di parentesi quadre e punti esclamativi. Riprendendo l'esempio precedente ed utilizzando le regole descritte si ottiene il risultato descritto dalla *figura 4*.

```
type nome_type1 {
    nome_field1 : tipo_di_dato
    nome_field2 : tipo_di_dato_non_nullable!
    nome_field3 : [array_di_dati]
    }
    type nome_type2 {
    nome_field1 : tipo_di_dato
    nome_field2 : tipo_di_dato_non_nullable!
    nome_field3 : nome_type1
    }
```

Figura 4 collegamento frai types

Esistono, anche i cosiddetti union *types* [1], questi creano un'unione tra due *types* in modo da avere tutte le informazioni insieme. Queste informazioni possono essere utilizzate, ad esempio, nelle *Query* o nelle *Mutation*. Per unire due o più *types* basta scrivere come è descritto in *figura 5*.

```
Union Nome_Unione = type1|type2|…|typen
```

Figura 5 definizione Union type

Infine, le *Query*[2] che le *Mutation*[3], sono considerate type, e come i precedenti vanno dichiarate nello schema [3]. All'interno di uno schema sotto il *type Query* vanno dichiarate tutte le Query che la nostra API sarà in grado di eseguire. Per far ciò basta scrivere il nome della *Query* come fosse un *field* e, a seguire, come nei *type* precedenti, il tipo di dato che essa deve dare come *return*[4]. Ci sono alcune cose che possiamo fare per rendere le Query più specifiche, ad esempio possiamo creare una *Query* che filtra i dati in base ad un *field*. Per far ciò basta semplicemente, subito dopo il nome della *Query*, tra parentesi tonde, scrivere un nome che identificherà il parametro di richiesta seguito dal tipo di dato da inserire successivamente. Le *Mutation*, a differenza degli altri *types*, necessitano di specificare che tipo di informazioni si vogliono avere in *input* ed anch'esse vanno specificate come espresso nelle *Query* con un filtraggio in base ad un parametro. Di seguito, rispettivamente, nella *figura 6* e nella *figura 7*, è espresso in maniera generica la dichiarazione di una *Query* e una *Mutation* in uno schema

```
type Query{
    Query1 : tipo_di_dato_in_output
    Query2 (parametri_filtro:tipo_dati_in_input) : tipo_di_dato_in_output
    }
```

Figura 6 definizione type Query

```
    Type Mutation {
    Mutation(input: tipo_dati_in_input): tipo_di_dato_in_output
    }
```

Figura 7 definizione type Mutation

Infine, esistono le *Subscription*, modalità che verrà discussa, ma non implementata nei *resolvers*, data la loro difficoltà di programmazione. Le *Subscription* sono richieste che un utente può fare, in cui si rimane in *ascolto* dell'API, e si aspetta che ci sia un cambiamento di un parametro richiesto. A questo punto l'API manda una notifica che specifica il cambiamento avvenuto [3]. Per l'implementazione delle *Subscription* nello schema basta scrivere ciò che è espresso nella *figura 8*.

```
type Subscription{
    nome_subscription: tipo_di_dato_in_output
}
```

Figura 8 definizione type Subscription

[2] Modalità per richiedere dati
[3] Modalità per modificare i dati
[4] Il dato che l'API da in ritorno alla richiesta

2.2.2 Query, Mutation e Subscription

Le *Query* e le *Mutation* sono protocolli utilizzati per richiedere dati (*fetching*) o modificarli [1], essi vengono dati in input ad un API che poi restituisce in uscita i dati richiesti, mentre le *Subscription* permettono di richiedere una notifica quando un particolare dato viene aggiunto o modificato. Le *Query*, *Mutation* e *Subscription* possibili devono essere precedentemente dichiarate nello schema. Per scrivere una *Query* bisogna scrivere "*Query*" (solo nelle operazioni di Query ciò può essere anche omesso[1]) e poi all'interno di *parentesi graffe* [5]inserire il nome della *Query* e se essa indica un *type* con più *fields*. All'interno di ulteriori *parentesi graffe* si possono inserire i *fields* che vogliamo richiedere. Se nello schema il dato in *output* non è un *type* è inutile fare una richiesta di *fields*. Ecco un semplice esempio di *Query* con i *fields* richiesti mostrato dalla *figura 9*.

```
1 query {
2     nome_query {
3         field1
4         field2
5     }
6 }
7
```

Figura 9 richiesta di Query in linguaggio GraphQL

Per le *Mutation* vanno inserite anche le informazioni in input [1]; ciò va fatto specificandolo tra *parentesi tonde*, si possono poi specificare i *fields* che si vogliono visualizzare in uscita come è espresso dalla *figura 10*.

[5] {

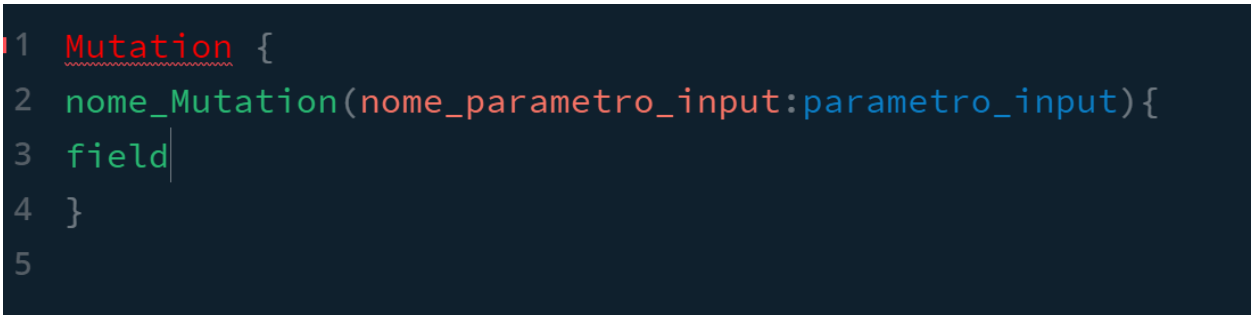

Figura 10 richiesta di Mutation in linguaggio GraphQL

Infine, le *Subscription* vanno richieste nella maniera descritta dalla *figura 11* [1].

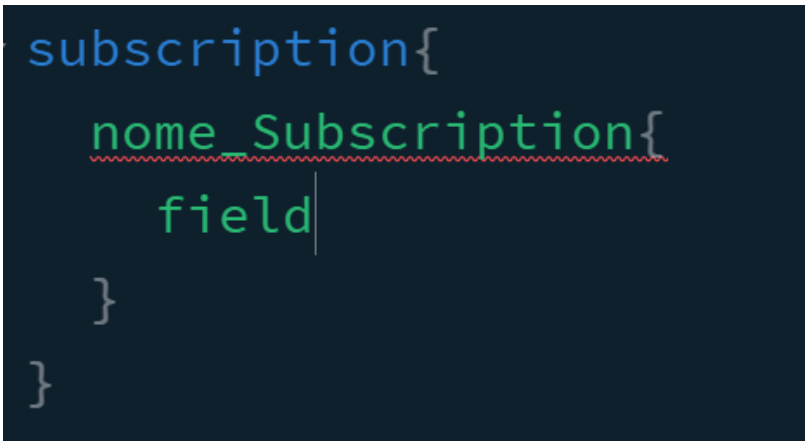

Figura 11 richiesta di Subscription in linguaggio Graphql

È possibile vedere a questo punto della tesi qualche esempio di *Query*, *Mutation* e *Subscription* applicate. Per queste prove sarà utilizzata come API, snowtooth moonhighway un API GraphQL sempre online, disponibile per fare esperienza del linguaggio GraphQL (disponibile al link: http://snowtooth.moonhighway.com[4]). Una semplice *Query* da cui iniziare è quella che richiede tutte la *Lift* dell'API, tramite la *Query AllLifts* che possiamo visualizzare con le altre *Query* all'interno della sezione *Docs* utilizzando questa *Query* e selezionando i *fields id*, *name* e *status* si otterrà il risultato descritto dalla *figura 12*.

Figura 12 Query AllLifts

Una possibile *Mutation*, invece, è quella di cambiare lo status di una *lift (SetLiftStatus)* [1], come possiamo vedere dalla *Query* precedente, la *lift* con *id "astra-express"* ha come stato *"CLOSED"* tramite la *Mutation* mostrata in *figura 13* è possibile modificare ciò.

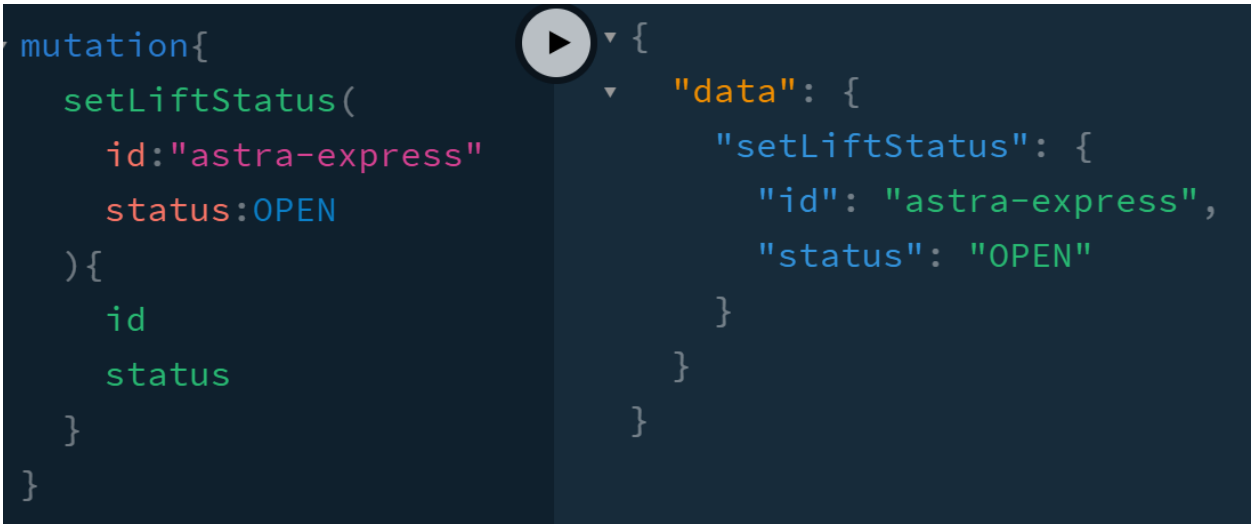

Figura 13 Mutation SetLiftStatus

Infine, è possibile vedere un esempio di una *Subscription*, si può richiedere di ricevere una notifica quando cambia lo *status* di una specifica *lift* nella seguente maniera mostrata in *figura 14*.

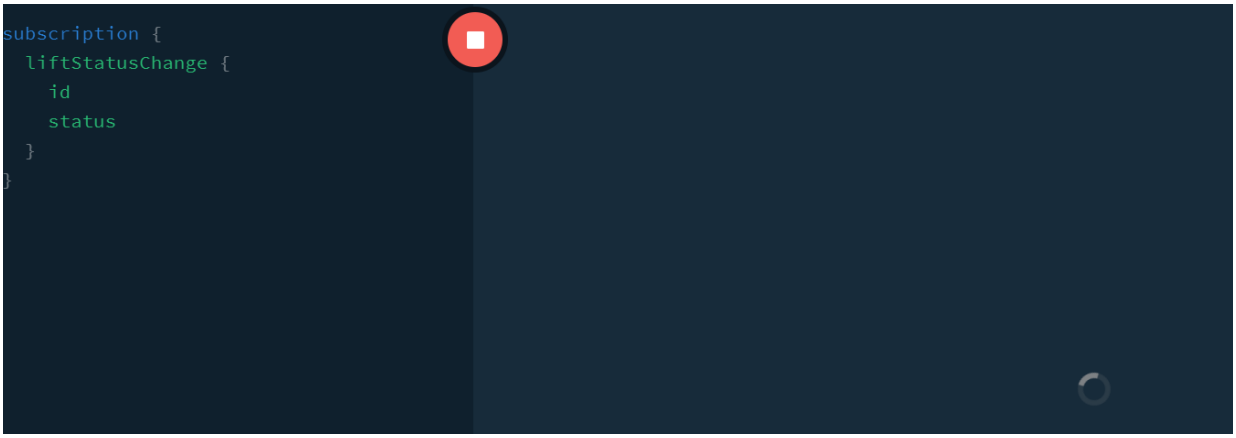

Figura 14 richiesta di Subscription

Come è possibile vedere la richiesta rimane attiva, poi in seguito ad una *Mutation* che cambia lo *status* di una *lift* comparirà ciò che è mostrato in *figura 15*.

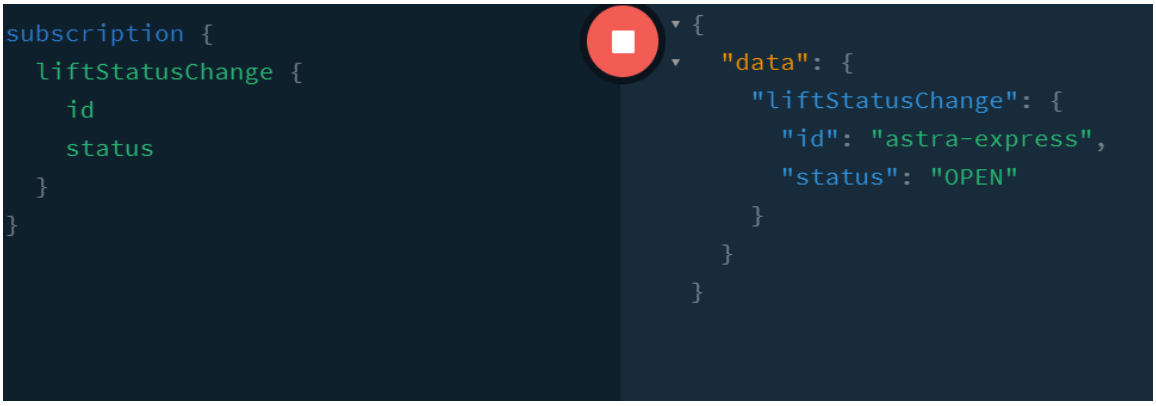

Figura 15 notifica di Subscription

2.2.3 Resolver

I *resolvers* sono ciò che implementa le Query e le Mutation descritte nello *schema* rendendole funzionanti. Esse sono funzioni e hanno principalmente quattro parametri [3]:

- Parent: Oggetto che contiene il risultato restituito dal resolver nel parent field.
- Args: è il parametro che l'utente fornisce in input all'API.

- Context: Questo oggetto è condiviso tra tutti i resolver che vengono eseguiti per una particolare operazione. Va utilizzato per condividere lo stato per una operazione, come le informazioni di autenticazione e l'accesso ai dati.
- Info: questo parametro utilizzato solo in API molto avanzate, contiene le informazioni relative all'avanzamento dell'operazione.

Questi parametri possono essere utilizzati tutti, solo alcuni o nessuno, come oggetti in ingresso ad una particolare funzione *resolver*. La particolarità dei *resolvers*, è che essi vengono scritti in maniera apposita in base alla funzione che si vuole dare all'API, e che devono essere scritte in base alla fonte di dati, quindi prendendo dati da database diversi, per creare una stessa funzione, si dovranno scrivere *resolvers* diversi.

2.3 Nodejs

Nodejs è un potente framework che permette la creazione di applicazioni web in *javascript* o *typescript*, linguaggio utilizzato nelle prove di questa tesi [5]. La piattaforma è basata sul *JavaScript Engine V8*, che è il *runtime*[6] di Google utilizzato anche da Chrome e disponibile sulle principali piattaforme [5]. La caratteristica più importante di *nodejs* è sicuramente il suo approccio asincrono, ovvero, permette di accedere alle risorse con una metodologia "*event- driven*", non utilizzando quindi il classico modello basato su *thread*[7] concorrenti utilizzati nei classici web server [5]. Il modello "*event-driven*" è abbastanza semplice: in sostanza un'azione viene lanciata solamente quando accade qualcosa. Ogni azione risulta quindi asincrona e l'attesa non dipende dal compimento di altre azioni richieste precedentemente. Grazie al comportamento asincrono, durante le attese di una certa azione il *runtime* può gestire, ad esempio, qualcos'altro che ha a che fare con la logica applicativa [6].

2.3.1 Nodemon

Nodemon è uno strumento che permette di utilizzare il comando *node* su un particolare codice ottenendo alcune funzionalità aggiuntive che permettono una maggiore comodità nella pratica della programmazione. La funzionalità probabilmente più efficace è quella che permette di lasciare l'API sempre online, aggiornandola e rifacendola partire solamente quando il codice viene aggiornato e in seguito salvato. Per utilizzarlo l'ho prima installato nell'applicazione attraverso il comando espresso in *figura 16* [7].

```
npm install nodemon
```

Figura 16 comando installazione della libreria nodemon

Poi all'interno del file *package.json*, alla dicitura *scripts*, si aggiunge il comando mostrato in *figura 17*.

[6] Runtime è il termine tecnico che indica il tempo in cui un programma viene eseguito [6]
[7] una suddivisione di un processo in due o più istanze o sottoprocessi che vengono eseguiti concorrentemente da un sistema di elaborazione monoprocessore (monothreading) o multiprocessore (multithreading) o multicore [6].

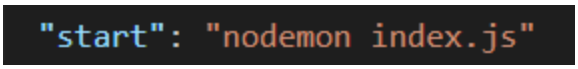

Figura 17 definizione comando start

dove *index.js* indica il file in cui sono indicate tutte le funzioni per far partire l'API. Inseguito quando il codice è pronto, basta scrivere nel terminale *npm start* e otterremo ciò che è mostrato in *figura 18*.

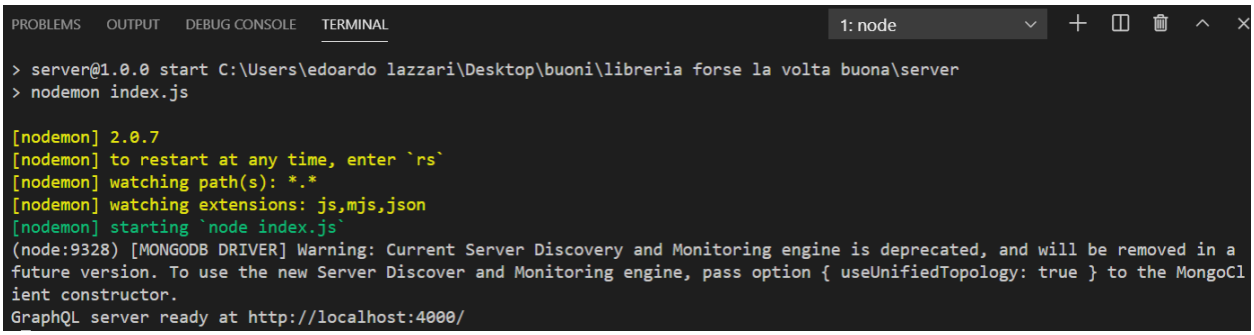

Figura 18 risposta del terminale al comando start

2.4 Apollo Server

Apollo Server è un server GraphQL open source gestito dalla comunità che lo utilizza. Funziona praticamente con tutti i *framework*[8] *server HTTP Node.js*. Apollo Server funziona con qualsiasi schema GraphQL utilizzando *SDL* (Schema Definition Language) [8].

Apollo Server si basa su alcuni principi [8]:

- È definito in base alle necessità dello sviluppatore.
- Viene mantenuto semplice per esser più facilmente implementato dalla comunità.

Per utilizzarlo, all'interno all'ambiente di sviluppo, basta semplicemente installarlo tramite il comando mostrato in *figura 19*.

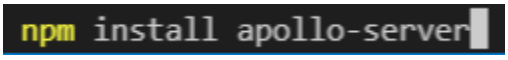

Figura 19 comando installazione della libreria apollo-server

Di seguito, *in figura 21*, un breve esempio di come viene utilizzato per avviare un semplice server GraphQL.

[8] è un'architettura logica di supporto sulla quale un software può essere progettato e realizzato [6]

```
const { ApolloServer, gql } = require('apollo-server');
const typeDefs = gql`
  type Query {
    hello: String
  }
`;
const resolvers = {
  Query: {
    hello: () => 'world',
  },
};
```

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
});
server.listen().then(({ url }) => {
  console.log(`🚀 Server ready at ${url}`);
});
```

Figura 20 esempio generale per la creazione di un server utilizzando ApolloServer

dove la costante *Typedefs* indica lo schema e viene passato, poi, insieme ai *resolvers*, nella creazione dell'API.

2.5 Mongodb

Mongodb è uno dei due database scelti in esame in questa tesi. È un database *NoSQL* orientato ai documenti che nasce nel 2007 in California e diventato, poi, un prodotto indipendente ed open-source [6] [9].

Con database *NoSQL* vengono definiti quei database "non relazionali", chiamati "*DB NoSQL*" o "non SQL" per evidenziare il fatto che sono in grado di gestire volumi elevati di dati non strutturati in rapida evoluzione in modi diversi rispetto a un database relazionale (*SQL*[9]) con righe e tabelle. Le tecnologie *NoSQL* sono disponibili fin dagli anni 1960, con nomi diversi, ma sono attualmente molto diffuse perché i dati sono in continua evoluzione e gli sviluppatori devono adeguarsi per riuscire a gestire il volume elevato e l'ampia varietà di dati generati da cloud, dispositivi mobili, social media e Big Data [6]. Mongodb memorizza i documenti in *JSON*[10], formato basato su *JavaScript* e più semplice di *XML*, ma comunque dotato di una buona espressività. In particolare, all'interno di questa tesi viene utilizzato MongodbCompass, programma installabile per utilizzare Mongodb come database. In questo caso

[9] Structured Query Language [6]
[10] JavaScript Object Notation [6]

l’apparenza del programma è mostrata in *figura 21*. Una volta cliccato Connect entreremo nel database dove potremmo visualizzare i vari databse dell’utente. Infine, cliccando su uno di essi potremmo visualizzare i dati come mostrato in *figura 22*.

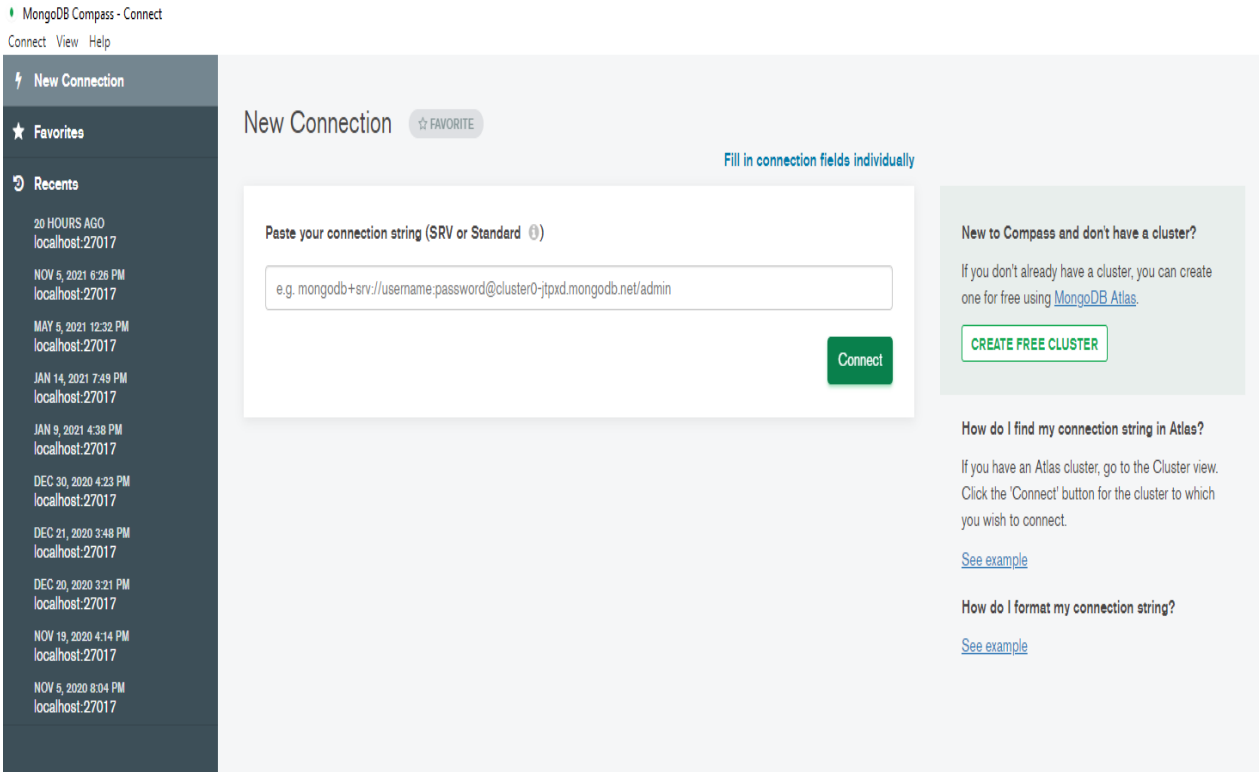

Figura 21 schermata iniziale MongodbCompass

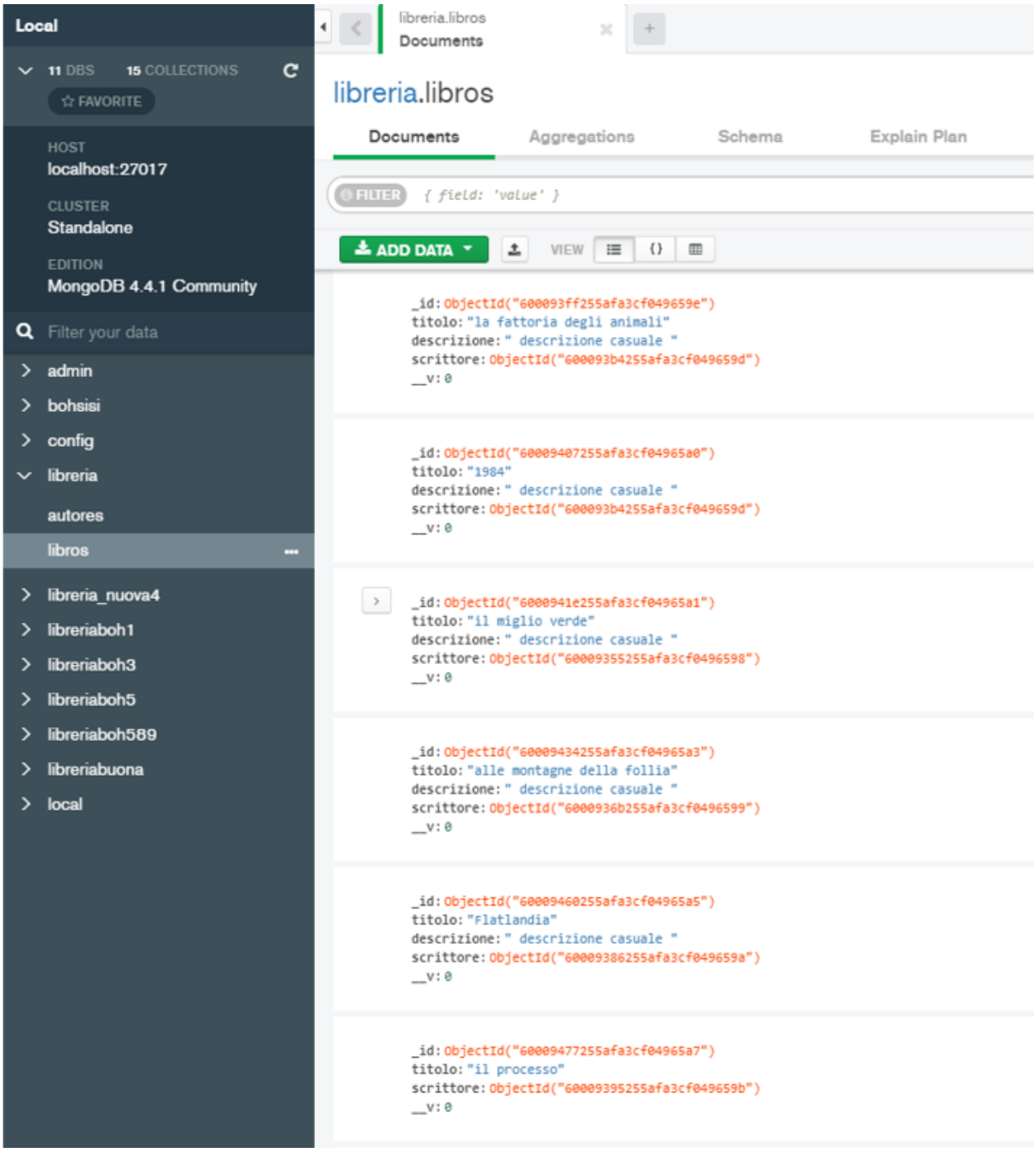

Figura 22 esempio generale database visualizzato in MongodbCompass

2.5.1 Mongoose

Moongose è una libreria, utilizzata in questa tesi, che permette di interagire coi database Mongodb consentendo di crearne, modificarli, eliminarli, in maniera facile e flessibile. L'installazione in ambienti di sviluppo è possibile tramite il comando descritto in *figura 23*.

```
npm install mongoose
```

Figura 23 comando installazione della libreria mongoose

Per creare un database si possono seguire le linee di codice scritte in *figura 24*.

```
4   const mongoose = require ('mongoose');
5   mongoose.connect('mongodb://localhost:27017/libreriaboh589', {
6       useNewUrlParser: true
7   });
```

Figura 24 esempio di connessione ad un database Mongodb

Ulteriori dettagli come la creazione dei documenti *JSON* e dei vari comandi per interagirci verranno analizzati in seguito durante la creazione dei *resolvers* [10].

2.6 Neo4j

Neo4j è un database a grafo open source. Anch'esso definito come database *noSql*, rientra però, a differenza del precedente, nella sottosezione di database a grafo (RDF[11]). Il modello di dati utilizzato con Neo4j è, quindi, un grafo, specificamente noto come *property graph* [11], e ci consente di modellare, memorizzare e interrogare i nostri dati come un grafo. I database a grafo come Neo4j sono ottimizzati per funzionare con dati strutturati a forma di grafo e l'esecuzione di attraversamenti complessi del grafo. Uno dei vantaggi dell'utilizzo di un database a grafo con GraphQL è che si mantiene lo stesso modello di dati in tutto lo stack di applicazioni, lavorando con database a grafo su entrambi il *frontend*[12], l'API e il *backend*[13]. Un altro vantaggio ha a che fare con le ottimizzazioni delle prestazioni ottenute dai database a grafo rispetto ad altri sistemi di database, come i database relazionali. Molte *Query* GraphQL finiscono per essere annidate a diversi livelli, l'equivalente di un file operazione *JOIN* in un database relazionale. I database a grafo sono ottimizzati per eseguire queste operazioni di attraversamento del grafo in modo efficiente; quindi, un database a grafo è una soluzione naturale [11].

[11] Resource Description Framework [6]
[12] Parte del programma con cui interagisce l'utente (interfaccia) [6]
[13] Parte del programma fa funzionare le interazioni dell'utente [6]

2.6.1 @neo4j/graphql

@neo4j/graphql è una libreria *Node.js* che funziona con qualsiasi implementazione *JavaScript* GraphQL, come Apollo Server, progettata per semplificare il più possibile la creazione di API GraphQL supportate da un database Neo4j. La funzione principale di questa libreria è creare *resolvers* dato uno *schema*: essa accetta le *type definition* (*schema* senza *Mutation* o *Query*) di GraphQL e genera un API GraphQL con operazioni *CRUD* (Create, Read, Update, Delete) per i *Types* definiti. Nella semantica GraphQL, ciò include l'aggiunta di un tipo di *Query* e *Mutation* allo schema e la generazione di *resolver* per queste *Query* e Mutation. L'API generata include il supporto per il filtraggio, l'ordinamento e l'impaginazione. Il risultato di questa funzione è uno schema da passare ad uno strumento, come apollo server per creare un'API funzionante [12].

2.6.2 GraphQL Architect

GraphQL Architect,è un'applicazione interna al database Neo4j che, una volta installata, permette di scrivere in un ambiente di sviluppo proprio solo uno schema GraphQL. Fatto ciò, essa crea automaticamente l'API, risparmiando all'utente la parte di installazione di pacchetti nell'ambiente di sviluppo e la creazione dell'API tramite Apollo Server. Di seguito è possibile vedere un esempio della schermata di GraphQL Architect in *figura 25* [11].

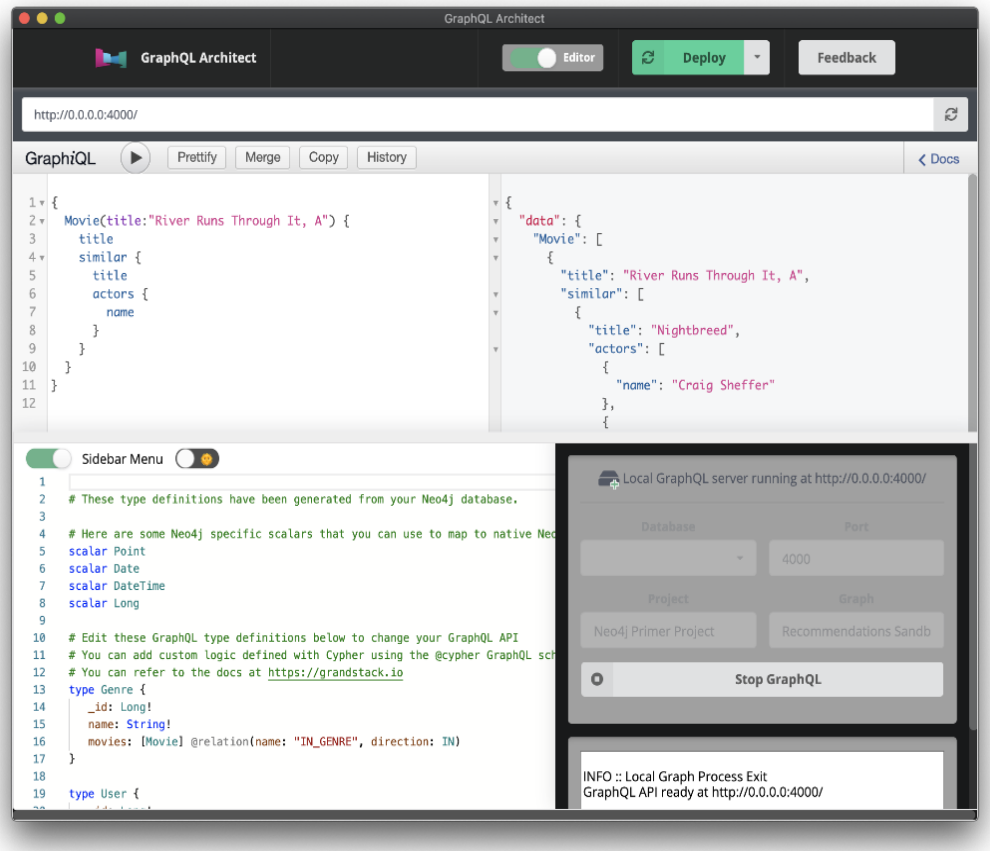

Figura 25 esempio schermata di GraphQL Architect

3 Sperimentazione: scrittura di una API basata su GraphQL utilizzando i due diversi database

In questa parte di tesi verrà mostrato il procedimento per la creazione di una API basata su GraphQL, utilizzando due diversi database, iniziando con le parti in comune per poi spostarsi sulle differenze.

3.1 Struttura comune ai due database

La parte comune alle due APIs, sarà, l'index, ovvero la parte del programma che crea il server, e lo schema che abbiamo definito precedentemente.

3.1.1 Creazione dell'index

Per prima cosa andrà creata una cartella che conterrà l'API. Aperta poi questa cartella all'interno dell'ambiente di sviluppo, tramite un terminale per la scrittura di comandi, verrà inizializzato il programma tramite il comando "npm init -y". Si procede con la creazione di index.js, l'indice appunto del nostro programma dove verrà creato il server in sé. Si installano poi i pacchetti utilizzati per entrambe le APIs, ovvero Apollo-server e nodemon, nelle modalità indicate precedentemente. A seguire si passa a scrivere l'index. La sua forma base, comune alle due APIs, sarà la seguente descritta in *figura 26*.

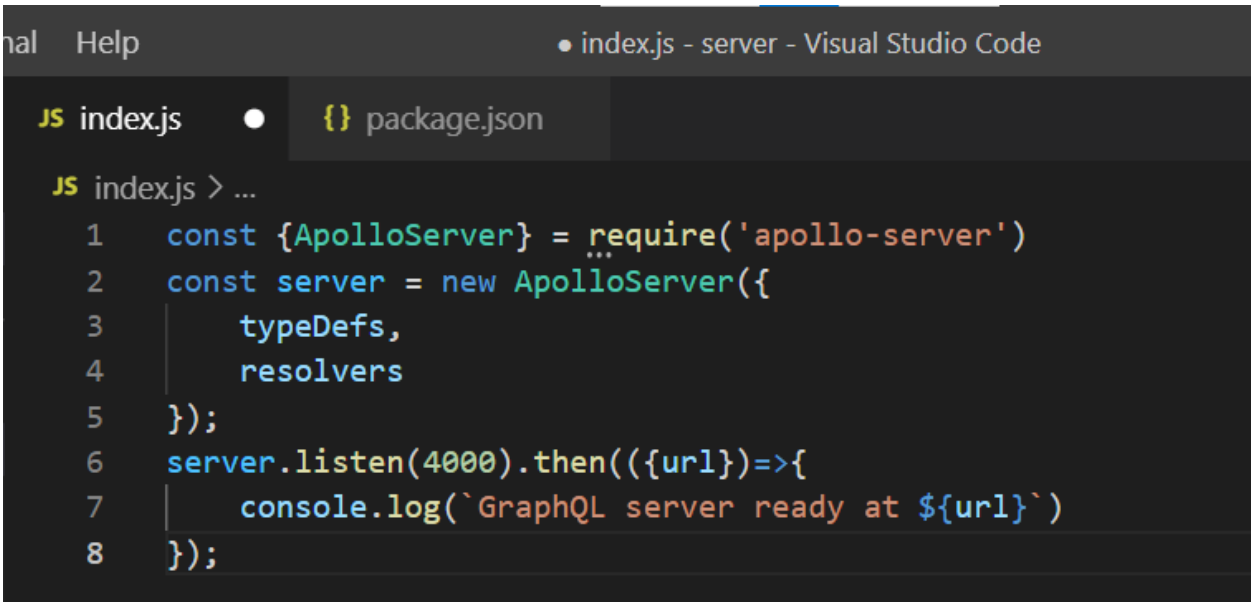

Figura 26 index generale

typeDefs indica la definizione dei *type*, ovvero lo *schema*, che poi verranno passati all'applicazione.

3.1.2 Scrittura schema

Per avere il codice più organizzato si crea innanzitutto una cartella GraphQL, in cui inserire in maniera ordinata tutto il codice. Dentro di essa verranno creati, poi una sottocartella *schema* al cui interno ci sarà il file *typeDefs.js* dove verrà scritto *schema*. Per averlo all'interno dell'*index* basta aggiungere la linea di codice:

const typeDefs= require ('./graphql/schema/typeDefs);

dove tra parentesi è indicato il percorso per identificare il file al cui interno c'è lo schema.

Passiamo quindi nel file *typeDefs.js* per scrivere lo *schema* che sarà comune alle due APIs, definito come segue in *figura 27*.

```
const {gql} = require ('apollo-server')
module.exports =gql`
type Libro{
    _id: ID!
    titolo: String!
    descrizione: String
    scrittore: Autore!
}
type Autore {
    _id: ID!
    nome: String!
    libri_scritti: [Libro!]
}
input Libroinput{
    titolo: String!
    descrizione: String
    scrittore : String!
}
input Autoreinput{
    nome: String!
}
type Query {
    tutti_libri: [Libro!]!
    autori: [Autore!]!
}
type Mutation{
    aggiungi_libro(libroinput: Libroinput) : Libro
    aggiungi_autore(autoreinput:Autoreinput) : Autore
}
 schema {
     query: Query
     mutation: Mutation
 }
`;
```

Figura 27 schema comune alle due APIs

3.2 Resolvers e loro componenti

Passiamo ora alle parti che differenziano le due APIs. La cosa importante da comprendere è come, in questa parte, nella scrittura dei *resolvers*, abbiamo la vera difficoltà di programmazione, dato che i *resolvers*, per la loro scrittura dipendono da molti fattori; le principali difficoltà in questa parte sono:

- la disponibilità di librerie per l'interazione con le varie fonti di dati: nello studio di questa tesi sono stati molti i database valutati per sviluppare le prove, ma la maggior parte di loro, soprattutto quelli molto nuovi e ancora in fase di sviluppo, spesso non dispongono di librerie molto ampie in grado di rendere facile la comunicazione database-programma, questo problema sarà sicuramente risolto in futuro con il lavoro che i vari sviluppatori fanno ogni giorno nell'aggiornamento di librerie e software.
- L'esperienza nella programmazione: da neofita di questo tipo di programmazione, son riuscito a ottenere qualche risultato, ma sono sicuro che con più esperienza nell'ambito si possano ottenere risultati più consistenti, in particolare penso sia interessante lo sviluppo di API in cui più fonti di interazione lavorano contemporaneamente, durante lo studio sono riuscito a sviluppare un progetto ibrido (reperibile al link: https://github.com/edlazzari8/mongoose-neo4j-graphql) in cui i due database utilizzati (Mongodb, Neo4j) lavorano contemporaneamente, ma purtroppo solo su chiamate distinte, sarebbe interessante, quindi, uno sviluppo di *resolvers* permettono con una chiamata unica l'interazione con entrambi i database.

3.2.1 Mongodb

Il codice completo di questa API può essere reperito al seguente link:
https://github.com/edlazzari8/mongoose-GraphQL

La prima cosa da fare, utilizzando Mongodb, è scegliere una libreria che consente di interagire con il database, in questa tesi è stata scelta moongoose. Questa libreria va installata come precedentemente illustrato, e aggiunto all'*index* la connessione col database come mostrato. È interessante notare che scegliendo il nome del database, se esso esiste già viene utilizzato, altrimenti viene creato in automatico.

3.2.1.1 Costruzione dei modelli

Essendo Mongodb un database *noSQL* basato su documenti *JSON* è necessario creare dei *modelli* che verranno utilizzati dal database per conoscere la struttura dei dati: avendo già uno *schema* è possibile basarsi sulla struttura già illustrata in esso.

Per prima cosa creiamo, all'interno della cartella dell'API, una sottocartella che conterrà i *modelli*, e all'interno di quest'ultima creiamo file .js all'interno dei quali scriviamo i *modelli* nel modo descritto dal codice in *figura 28*.

```
const mongoose = require ('mongoose');
const Schema = mongoose.Schema;
const libroSchema = new Schema({
    titolo: {
        type: String,
        required: true
    },
    descrizione: {
        type : String,
        required : false
    },
    scrittore : {
        type : Schema.Types.ObjectId,
        ref : 'Autore'
    }
});
module.exports = mongoose.model('Libro', libroSchema)
```

Figura 28 costruzione di un modello

È possibile vedere come all'interno della costruzione dello *schema*, nel *field libri_scritti*, utilizziamo una funzione di mongoose per collegarlo ad un altro *type*, che viene poi specificato con *ref*, infine viene esportato il modello per essere utilizzato in altri parti del codice dove è possibile richiederlo.

3.2.1.2 Creazione dei resolvers

Per la scrittura dei resolvers delle *Query* e delle *Mutation* della nostra API, utilizzeremo per la maggior parte funzioni all'interno della libreria Mongoose. Tutta la documentazione sulle funzioni si può trovare al link GitHub - edlazzari8/mongoose-GraphQL [10].

Faremo solo un'eccezione per permettere le richieste "*nested*[14]". Al posto di utilizzare la funzione *populate()* che permette di inserire le informazioni di un modello all'interno dell'altro, creeremo delle nostre funzioni che faranno ciò. Il motivo di questa scelta è il seguente: la funzione *populate()* può esser chiamata più volte riempiendo ogni volta il modello con le informazioni dell'altro questo anche in maniera sequenziale potendo creare dei *loop* fastidiosi, mentre tramite le funzioni *create* seguentemente, ogni modello verrà riempito di informazioni, singolarmente solo alla chiamata della funzione. Le funzioni sopracitate, scritte all'interno di un file denominato *Merge.js*, sono le seguenti esplicitate in *figura 29*.

[14] annidate

```
const autore = async autoreId => {
    try {
        const autore = await Autore.findById(autoreId);
        return {
            ...autore._doc,
            _id: autore._doc._id.toString(),
            libri_scritti: libri.bind(this, autore._doc.libri_scritti)
        };
    } catch (err) {
        throw err;
    }
};
const trasformalibro = libro => {
    return {
        ...libro._doc,
        _id: libro._doc._id.toString(),
        scrittore: autore.bind(this, libro._doc.scrittore)
    };
};
exports.trasformalibro = trasformalibro;
exports.autore = autore;
exports.libri = libri;
exports.librosingolo = librosingolo;
```

```
const Libro = require('../../modelli/libro')
const Autore = require('../../modelli/autore')
const libri = async libriIds => {
    try {
        const libri = await Libro.find({ _id: { $in: libriIds } });
        return libri.map(libro => {
            return trasformalibro(libro);
        })
    } catch (err) {
        throw err;
    }
};
const librosingolo = async libroId => {
    try {
        const libro = await libro.findById(libroId);
        return trasformalibro(libro);
    } catch (err) {
        throw err;
    }
}
```

Figura 29 funzioni merge

Possiamo passare quindi alla scrittura dei *resolvers*, questi andranno divisi in base al loro esser *Query* o *Mutation* nella seguente maniera mostrata in *figura 30*.

```
Module.exports = {
    Query:{
        Resolver_query()
    },
    Mutation :{
        Resolver_mutation()
    }
}
```

Figura 30 schema generale resolvers

Partiamo dai *resolvers* delle *Query* che sono in questo caso più semplici. Nello schema abbiamo definito due *Query*, *tutti_libri: [Libro!]!* e *autori: [Autore!]!*, che semplicemente richiedono i libri e gli autori inseriti nel database.

Di seguito il codice che li implementa in *figura 31*.

```
tutti_libri: async() => {
    try {
        const libri = await Libro.find();
        return libri.map(libro =>{
        return merge.trasformalibro(libro);

        });
    } catch (err) {
        throw err;
    }

},
autori: async () => {
    try {
        const autore = await Autore.find();
        return autore.map(autore => {
            return {
                ...autore._doc,
                _id: autore._doc._id.toString(),
                libri_scritti: merge.libri.bind(this, autore._doc.libri_scritti)
            };
        });
    } catch (err) {
        throw err;
    }

}
```

Figura 31 resolvers delle Query

Come possiamo vedere le funzioni sono divise in *try catch* e *throw*: essendo funzioni asincrone ciò permette di evitare errori inconvenienti. Analizzandone una delle due, ad esempio *tutti_libri*, possiamo vedere come all'inizio vengono richiesti al database tutti i dati del *modello libro*, poi tramite una delle funzioni descritte in precedenza vengono forniti all'utente contenenti anche le informazioni del modello autore.

Passiamo ora ai *resolver* un po' più complicati, le *Mutation*, che non devono solo richiedere dati ma anche modificarne o aggiungerne. In questa API ne abbiamo due, uno che crea un'entità *Autore*, e uno che crea un'entità *Libro*. Per scelte di programmazione, ho impostato le funzioni in maniera che con la creazione di *Autore* si crei solo l'entità *Autore*, mentre il collegamento libro autore viene generato con la funzione che crea un *libro*, rendendo quest'ultima leggermente più complicata. Vediamo prima la creazione di autore implementata dal codice in *figura 32*

```
aggiungi_autore: async (parent, args, context, info) => {
    try {
        const autore_già_inserito = await Autore.findOne({ nome: args.autoreinput.nome });
        if (autore_già_inserito) {
            throw new Error('autore già inserito');
        }
        const autore = new Autore({
            nome: args.autoreinput.nome
        });
        const result = await autore.save();
        return { ...result._doc, _id: result._doc._id.toString() };


    } catch (err) {
        throw err;
    }

},
```

Figura 32 resolver aggiungi_autore

Possiamo vedere innanzi tutto, come è utilizzato il parametro *args*, precedente definito come ciò che viene dato in input. Infatti, all'interno di *args* abbiamo in ordine prima *autoreinput*, che contiene a sua volta il nome. Per avere quindi il nome scriviamo *args.autoreinput.nome*. Successivamente, possiamo vedere un piccolo controllo finalizzato all'evitare ridondanza di dati nel database, facendo un controllo se l'autore è già presente nel database, in caso non sia presente viene inserito e viene poi dato in *return* dato che la *Mutation*, per sua definizione nello schema fornisce in *output* i dati dell'autore. Di seguito si propone un esempio per mostrare come aggiungere un libro al cui interno viene formata la connessione con l'*Autore* come mostrato in *figura 33*.

```
aggiungi_libro: async(parent, args, context,info) =>{
    try {
        const autore_non_inserito = await Autore.findOne({ nome: args.libroinput.scrittore });
        if (!autore_non_inserito) {
            throw new Error('autore non inserito')
        }
    } catch (err) {
        throw err;
    }
    const libro = new Libro({
        titolo: args.libroinput.titolo,
        descrizione: args.libroinput.descrizione,
        scrittore: await Autore.findOne({ nome: args.libroinput.scrittore })
    });
    let librocreato;
    try {
        const result = await libro.save();
        librocreato = merge.trasformalibro(result)
        const scrittore = await Autore.findOne({ nome: args.libroinput.scrittore })
        scrittore.libri_scritti.push(libro);
        await scrittore.save();
        return librocreato;
    } catch (err) {
        throw err;
    }
},
```

Figura 33 resolver aggiungi_libro

Inizialmente si esegue un controllo per vedere se è già presente l'autore nel database: dato che nello *schema* abbiamo messo "*Autore*" come un parametro che non può esser nullo, se l'*autore* non è presente nel database viene mandato in output un errore, altrimenti viene creato il libro e collegato con l'*autore*.

3.2.2 Neo4j

Vediamo di seguito la metodologia per la costruzione di *resolvers* utilizzando come database Neo4j, il codice completo per quest'API può essere reperito al seguente link: https://github.com/edlazzari8/neo4j-GraphQL

3.2.2.1 Impostazioni per la realizzazione di resolvers automatici

Le prime impostazioni preliminari sono alcuni passaggi come l'installazione dei pacchetti e la connessione al database. I pacchetti utilizzati sono neo4j-driver, che permette di comunicare col database e @neo4j/graphql, utile per creare *resolvers* automaticamente. I comandi per installarli sono identici a quelli esplicitati precedentemente, di seguito espresso in *figura 34* [12].

```
npm install @neo4j/graphql neo4j-driver
```

Figura 34 comandi installazione pacchetti neo4j

All'interno dell'*index* dobbiamo creare il collegamento con il database e impostare poi lo schema dal quale la libreria creerà i *resolver*. Successivamente bisognerà passare tutto ciò ad Apollo Server. Questa azione è resa possibile, all'interno dell'*index*, tramite il codice in *figura 35* [12].

```
const { ApolloServer } = require ("apollo-server");
const neo4j = require("neo4j-driver");
const { makeAugmentedSchema, neo4jgraphql } = require ("neo4j-graphql-js");
const typeDefs = require('./graphql/schema')
const resolvers = require('./graphql/resolvers')
const schema = makeAugmentedSchema({
    typeDefs,
    resolvers
});
```

```
const driver = neo4j.driver(
    "bolt://localhost:7687",
    neo4j.auth.basic("prova", "password")
);
const server = new ApolloServer({
    schema,
    context: { driver }
   });

server.listen().then(({ url }) => {
    console.log(`GraphQL server ready at ${url}`);
});
```

Figura 35 impostazioni preliminari per neo4j

In seguito, basterà passare come *typedefs* uno schema GraphQL leggermente modificato: quando c'è un collegamento tra due *type* bisogna specificare il nome del collegamento e se è entrante o uscente dal *type*. Un esempio è lo schema base a cui è stata tolta la parte di *Query* e *Mutation* in quanto quest'ultime verranno generate dalla libreria come mostrato in *figura 36*.

```
const {gql} = require('apollo-server')
module.exports =gql`
type Libro {
    titolo : String!
    descrizione : String
    autore: Autore @relation(name:"STATO_SCRITTO_DA", direction : OUT )
}
type Autore {

    nome: String!
    libri_scritti: [Libro!] @relation(name:"HA_SCRITTO", direction : OUT )
}
```

Figura 36 schema utilizzato con neo4j

Con il comando *npm start* è possibile poi avviare la nostra Api e vedere all'interno dei *docs*[15] le *Query* e le *Mutation* create dalla libreria apposita. Ecco un esempio in *figura 37*.

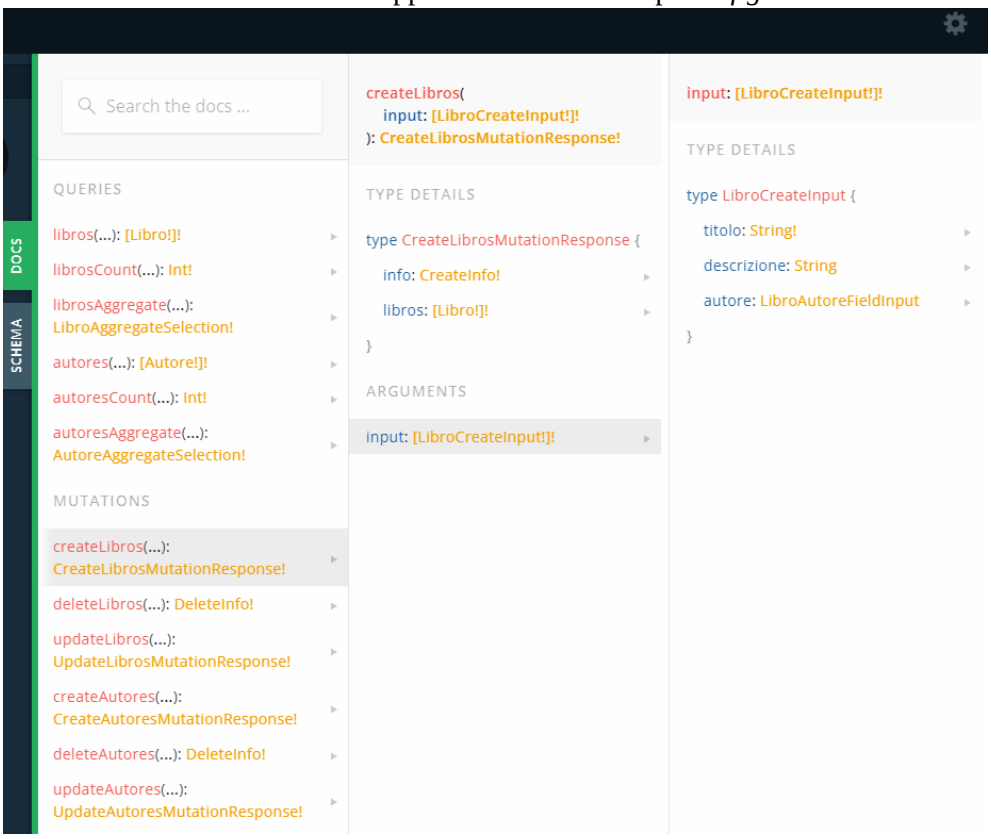

Figura 37 docs di Query e Mutation autogenerate

[15] documenti

3.2.2.2 Scrittura di resolvers custom

È possibile anche scrivere dei resolvers personalizzati cosicché le richieste fatte rispondano precisamente allo scopo prefissato. Un modo è aggiungere nello schema l'istruzione in linguaggio *cypher* (il linguaggio nativo di Neo4j). Un esempio con le *Query* e le *Mutation* che erano state create precedentemente utilizzando Mongodb (capitolo 3.2.1) è mostrato di seguito in *figura 38*. [12]

```
input Libroinput{
    titolo: String!
    descrizione: String!
    autore : String!
}
input Autoreinput{
    nome: String!
}
```

```
type Query {
    tutti_libri : [Libro] @cypher(
        statement:"""
        MATCH (a:Libro) - [:STATO_SCRITTO_DA] -> (b:Autore)
        return a,b
    """)
    autori : [Autore] @cypher(
        statement:"""
        MATCH (a:Autore) - [:HA_SCRITTO] -> (b:Libro)
        return (a),(b)
    """)
}
```

```
type Mutation {
    aggiungi_libro(libroinput: Libroinput) :Libro @cypher(
        statement:"""
        MERGE (a:Libro {titolo: $libroinput.titolo, descrizione: $libroinput.descrizione})
        MERGE (b:Autore {nome: $libroinput.autore})
        MERGE (a) <- [:HA_SCRITTO] - (b)
        MERGE (b) <- [:STATO_SCRITTO_DA] - (a)
        return a,b
    """)
    aggiungi_autore(autoreinput: Autoreinput) : Autore @cypher(
        statement:"""
        MERGE (b:Autore {nome: $autoreinput.nome})
        return (b)
    """)

}
```

Figura 38 scrittura di resolvers personalizzati

Utilizzando questa modalità, in cui specifichiamo le istruzioni in linguaggio *cypher* possiamo ampliare le possibilità offerte dai *resolvers* creati dalla libreria.

4 Conclusioni

Come abbiamo visto GraphQL è un potente strumento che si può utilizzare per scrivere APIs flessibili, in grado di far comunicare vari fonti di informazioni, le quali singolarmente possono anche comunicare con linguaggi diversi. La problematica maggiormente riscontrata durante la scrittura dei *resolvers*. Tuttavia, come mostrato si possono ottenere risultati accettabili anche per neofiti nel linguaggio di programmazione utilizzato. Infatti, utilizzando i due programmi descritti precedentemente è possibile creare anche un'unica API sulla quale fare distinte richieste ad un database piuttosto che ad un altro ottenendo informazioni singolarmente da entrambi. Un futuro step interessante per questo progetto potrebbe essere scrivere *resolvers* che permettano, tramite una sola richiesta, il *fetching* o la modifica di entrambi i database contemporaneamente. Ciò potrebbe essere possibile tramite nuove librerie e maggiori esperienze di programmazione dell'utente sull'argomento. In conclusione, con l'uscita e l'aggiornamento di sempre nuove di librerie sarà possibile scrivere in maniera sempre più semplice APIs flessibili grazie a GraphQL.

Link al codice

Di seguito sono disponibili i link al codice completo dei delle due API separate

https://github.com/edlazzari8/mongoose-GraphQL

https://github.com/edlazzari8/neo4j-GraphQL

l'ultimo invece riguarda il codice ibrido citato:

https://github.com/edlazzari8/mongoose-neo4j-graphql

Fonti

[1]. Learning GraphQL Eve Porcello & Alex Banks

[2]. GraphQL playground, https://github.com/graphql/graphql-playground

[3]. Graphql, https://graphql.org/

[4]. Snowtooth moonhighway (API GraphQL pubblica), http://snowtooth.moonhighway.com

[5]. NodeJs, https://nodejs.org/en/

[6]. Wikipedia, https://it.wikipedia.org/

[7]. Nodemon, https://nodemon.io/

[8]. Apollo-Server (docs del sito), https://www.apollographql.com/docs/apollo-server/

[9]. Mongodb, https://www.mongodb.com/it-it

[10]. Mongoose (docs), https://mongoosejs.com/docs/guide.html

[11]. Neo4j, https://neo4j.com/

[12]. Neo4j (documenti libreria per interagire con GraphQL), https://neo4j.com/docs/graphql-manual/current/