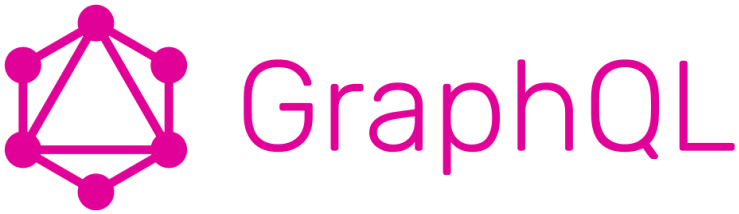
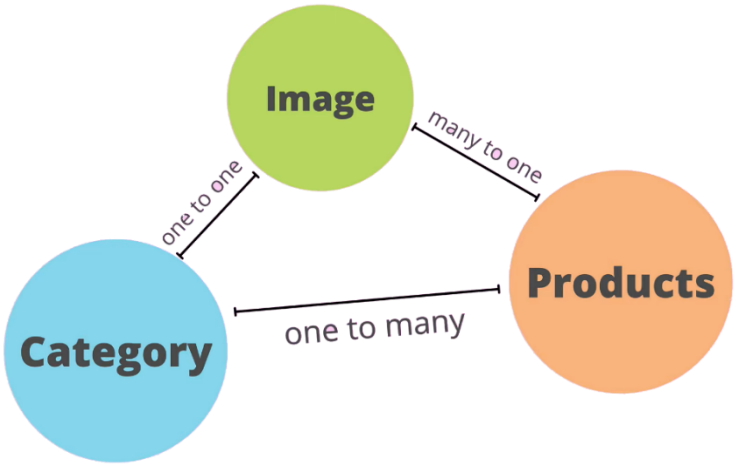
Corso GraphQL



Cap.1 - Le Basi - Parte 1

- **Cos'è GraphQL?**

GraphQl è un query language strutturato a **grafo** che funge da middleware tra FrontEnd e BackEnd



Al pari delle API REST definisce delle tipologie di gestione dell'informazione.

API REST (Request)	GraphQL (Types)
GET	Query
PUT	Mutation
DELETE	Mutation
POST	Mutation

GraphQL può essere utilizzato in **due modi**:

1. Come middleware tra il FE e il BE
2. Come vero e proprio servizio costituente il BE (express API con gQL integrato all'interno)

La seconda opzione è quella da utilizzare per maggiore immediatezza.

Cap.2 - Le Basi - Parte 2

- **Tipi di dato**

In **gQL** esistono **cinque tipi** di **dato (Scalars)**:

- ID
- String
- Int
- Float
- Boolean

L'obbligatorietà del campo richiesto è definita dalla giustapposizione del **nullable (!)**

GraphQL definisce uno **schema**, ovvero una struttura che definisce tutti i dati presenti nella tua **API**.

Ogni **gQL** Schema ha una **root type** chiamata **Query**.

Es1. di query che richiama una lista di utenti

Nota: è possiible strutturare la query lato client anche in modo **anonimo**, inserendo al'interno le Query create.

```
type User {
  id: ID!
  name: String!
  age: Int!
  heigth: Float!
  isMarried: Boolean
  friends: [User]
  videosPosted: [Video]
}

type Video {
  id: ID!
  title: String
  descrption: String!
}
```

Figura 1 Struttura di dati concernente un utente che posta video e che ha altri utenti correlati.

```
type Query {
  users: [User]
  user(id: ID): User
}
```
OGGETTO
PARAMETRO
CHIAMATA

Facciamo un esempio di query…

Struttura:

Apertura di graffe con all'interno una query che richiama i dati.

Nota: Essendo la risposta non un tipo di dato primitivo

(`type Country {`) bisognerà aggiungere i parametri richiesti.

```
input userInput {
  id: ID
  name: String
}

type Query {
  user(input: userInput): User
```

Figura 2 Creazione di un input unitario per definire un gruppo di parametri

```
{
  country(code: "US") {
  code
  name
  phone
  capital
  currency
  }
}
```

…ecco la risposta!

```
{
  "data": {
    "country": {
      "code": "US",
      "name": "United States",
      "phone": "1",
      "capital": "Washington D.C.",
      "currency": "USD,USN,USS"
    }
  }
}
```

Il tutto a partire da questa root query presente nello schema

```
type Query {
  _entities(representations: [_Any!]!): [_Entity]!
  _service: _Service!
  countries(filter: CountryFilterInput): [Country!]!
  country(code: ID!): Country
```

Nota:

Se nella query fosse stato inserito il parametro **Continent**, essendo composto, avremmo dovuto specificare che parametri, a sua volta avremmo dovuto prendere.

```
    currency
      continent {
        code
        name
      }
    }
}
```

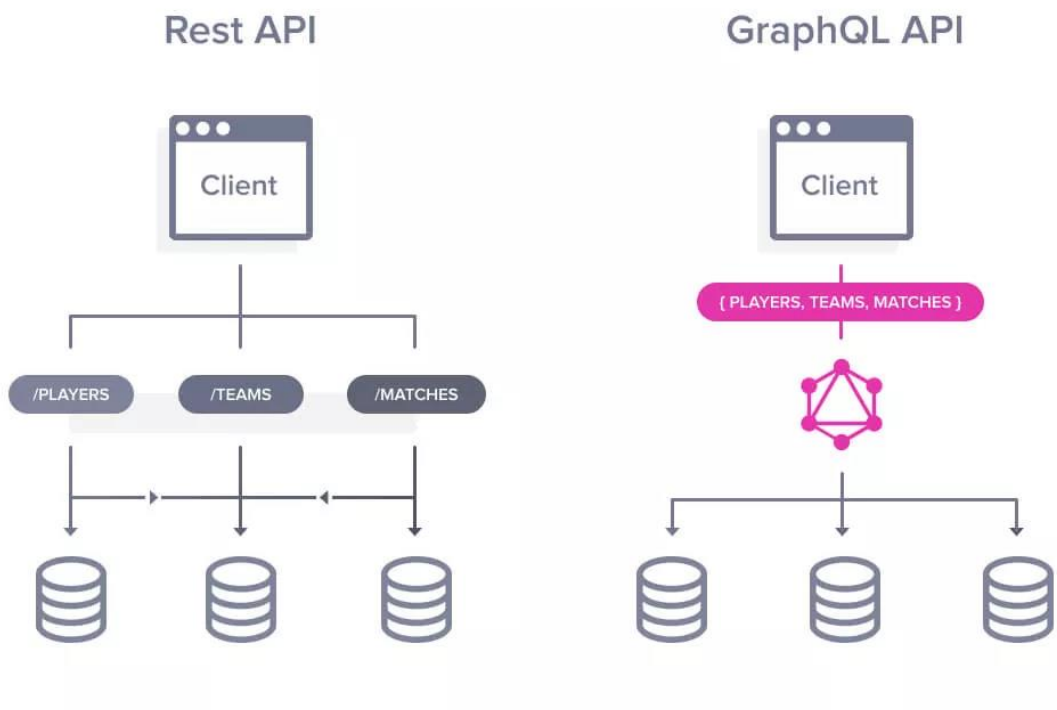


Figura 3 Comparazione tra API Test e GraphQL

*API free da utilizzare per testare GraphQL: https://github.com/machadop1407/graphql-full-course

Cap.3 – Setup e installazioni

1. npm init
2. Predisponi il progetto per typescript
   (vedi: https://www.digitalocean.com/community/tutorials/setting-up-a-node-project-with-typescript)
3. npm i apollo-server
4. npm i graphql

Non c'è bisogno di installare express!

5. Creazione dell'**index.js**
6. Import e istanziamento del **Server Apollo**

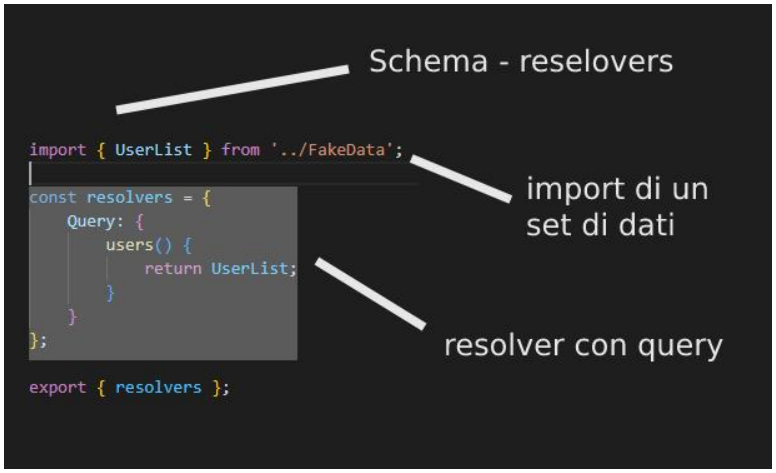
```
const server = new ApolloServer({typeDefs, resolvers});
```

`typeDefs:` Contiente tutti i tipi di dati e i **tipi di chiamate** (Query e Mutation)

`resolvers:` Contiene tutte le query e funzioni

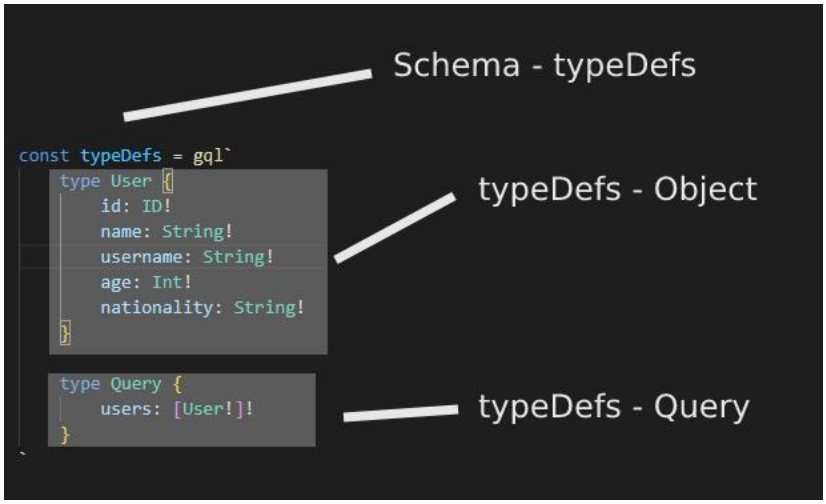
Insieme compongono lo **Schema**

7. Creazione della **cartella schema** di **gQL** per mantenere tutti i 2 files **resolvers** e **typeDefs**.
8. Creazione dei file **typedefs**



Definisce la **struttura dei dati** e delle eventuali query che verranno effettuate.

9. Creazione dei file **resolver**



Query volte a richiamare dati da un db.

Nel caso esempio i dati vengono tratti da un generico json.

10. Esempio di chiamata tramite l'API test integrato in Apollo

```
query ExampleQuery {
  users {
    id
    username
  }
}
```

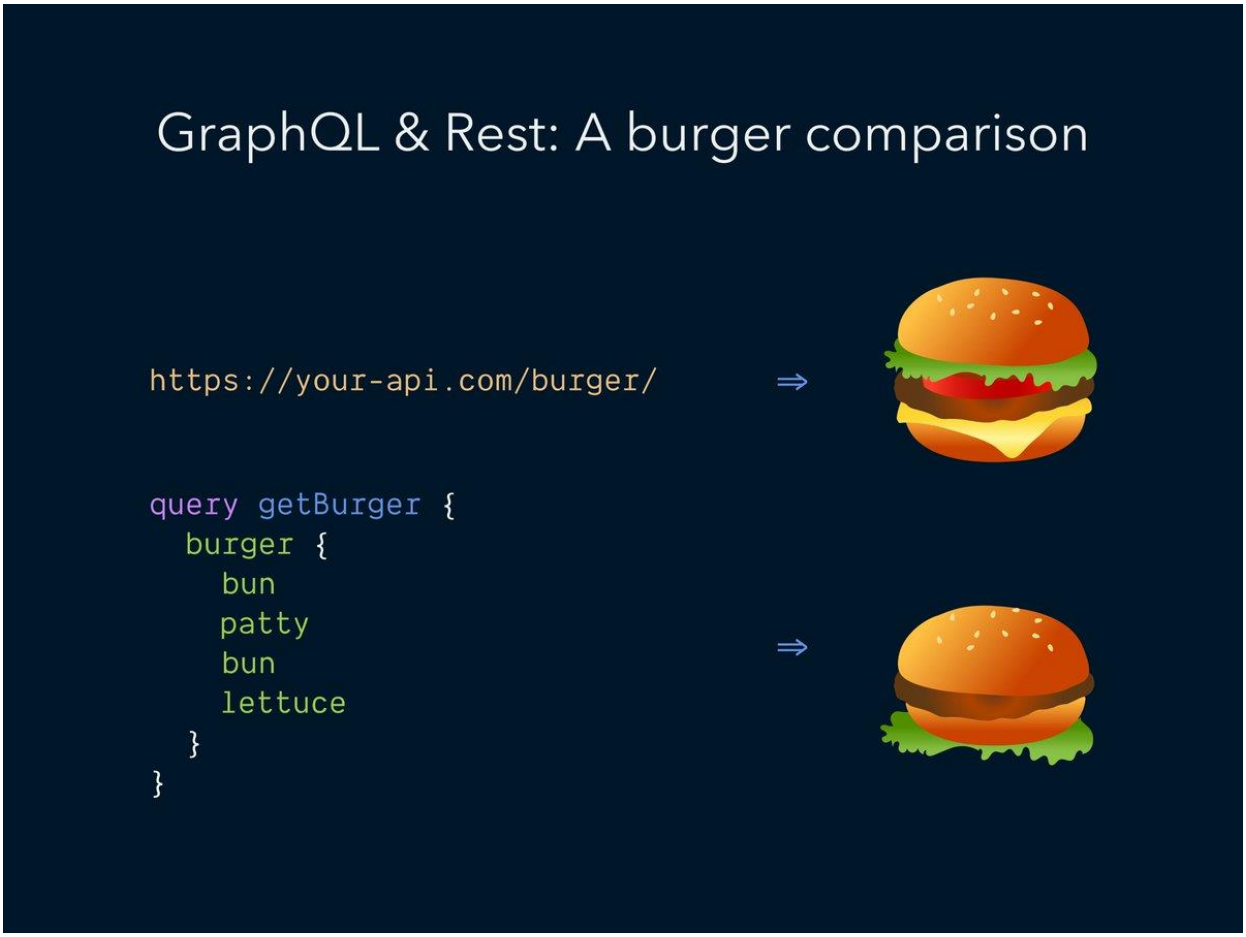
La query può avere qualsiasi nomenclatura, ma all'interno bisognerà definire il **tipo di query users** con i relativi campi di cui abbiamo bisogno.

11. Integriamo un **ORM specifico** (prisma) per trarre i dati da un DB già esistente
    - npm install prisma -D
    - npx prisma init
    - configuriamo file env
    - npx prisma db pull
    - npm install @prisma/client
    - npx prisma generate --schema prisma/schema.prisma
    - Instanziamo il client e importandolo nel resolver

```
import { PrismaClient } from '@prisma/client'
const prisma = new PrismaClient()

const resolvers = {
    Query: {
        users() {
            const UsersList = prisma.users.findMany({})
            return UsersList;
        }
    }
};

export { resolvers };
```

Nota: ricorda sempre di inserire il **notnullable** (**!**) solo quando si ha la certezza del tipo di dato corrisposto in chiamata nel db, altrimenti verrà dato un errore.

Nota 2: Istanziare i **typedefs/resolvers** senza contenuto porterà ad un errore.



File adibito alla creazione delle query con lo scopo di ottenere o modificare dati nel db.

Alle funzioni resolver vengono passati quattro argomenti: **parent**, **args**, **context** e **info** (in quest'ordine).

parent	**Il valore di ritorno del resolver per l'elemento padre di questo campo (ovvero, il resolver precedente nella catena di resolver). Per i resolver di campi di primo livello senza padre (come i campi di Query), questo valore viene ottenuto dalla funzione rootValue passata al costruttore di Apollo Server.**
. args	Un oggetto che contiene tutti gli argomenti GraphQL forniti per questo campo. Ad esempio, quando si esegue query{ user(id: "4") }, l'oggetto args passato al resolver utente è { "id": "4" }.
context	Un oggetto condiviso tra tutti i resolver in esecuzione per una particolare operazione. Usalo per condividere lo stato per operazione, incluse le

	informazioni di autenticazione, le istanze del caricatore di dati e qualsiasi altra cosa da tenere traccia tra i resolver.
info	Contiene informazioni sullo stato di esecuzione dell'operazione, incluso il nome del campo, il percorso del campo dalla radice e altro ancora.

Alcuni concetti:

- **Enum**: Le enum consentono a uno sviluppatore di definire un insieme di costanti denominate. L'uso delle enumerazioni può semplificare la documentazione dell'intento o creare un insieme di casi distinti. TypeScript fornisce enumerazioni sia numeriche che basate su stringhe.

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
```

Gli enum sono utili per definire una limitazione in più in fase di definizione dei dati, **enumerandoli**.

Nota: Ma attenzione, nulla al di fuori di essi (a parte i **null**) verrà concesso.

- **Relation:** relazione tra un oggetto del db ed un altro

```
type User {
    id: ID!
    name: String!
    username: String!
    age: Int!
    nationality: Nationality
    friends: [User]
    favoritesMovies: [Movie]
}

type Movie {
    id: Int!
    name: String!
    yearOfPublication: Int!
    isInTheaters: Boolean!
}
```

È possibile definire una relazione tra una tabella ed un'altra creando, nella sezione resolver dei query resolver adibiti a creare una relazione, questo permetterà di ottenere un pool di dati relazionati di 2 tabelle insieme

Figura 4 Un utente (tabella user) può avere più film preferiti (tabella movies)

```
const resolvers = {
    Query: {···
    },
    User: {
        favoritesMovies() {
            const moviesList = prisma.movies.findMany({})
            return moviesList;
        }
    }
};
```

Query resolver usato per le chiamate lineari

Query resolver "relazionale" usato per relazionale 2 tabelle distinte

Esempio di **flusso** di una **Query**, dal **BE** al **FE**

typeDefs

```
type Query {
    hello: String!
    utenti: [Utente]
}
```

resolvers

```
const resolvers = {
    Query: {
        hello: () => {
            return "Hello, Man!"
        },
        utenti: () => {
            return prisma.utente.findMany({})
        }
    }
};
```

client

```
query {
  hello
  utenti {
    id
    name
    username
  }
}
```

Esempio di **query** di un **array** di **oggetti**

typeDefs

```
type Products {
    id: ID!
    name: String!
    description: String!
    image: String!
    quantity: Int!
    price: Float!
    onSale: Boolean!
}
```

```
type Query {
    hello: String
    utenti: [Utente]
    products: [Products]
}
```

resolver

```
const resolvers = {
    Query: {
        products: () => {
            return data.products
        },
```

client

```
query Products {
  products {
    name
    price
    quantity
  }
}
```

Esempio di **query** a **singolo oggetto**

- **Query con associazione di tabelle**

1. **Relazione Genitore → Figlio**

È possibile associare più tabelle **nidificando** nello **schema** dei **typedefs** oggetti già esistenti e correlandoli facendo poi restituire i dati associati da un **resolver** specifico.

Prendiamo ad esempio una serie di prodotti, ognuno di essi apparterranno ad una categoria, i singoli prodotti saranno associati ad una categoria specifica da un **categoryId** insito nel singolo oggetto Prodotto.

Relazione: Categoria **1 ----> N** Prodotto

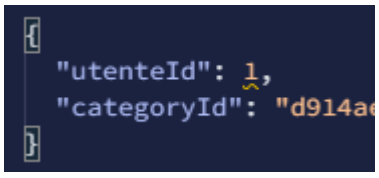
Stampando le categorie esse avranno una serie di prodotti appartenenti ad essa.

Figura 5 Flusso di chiamata di tabelle correlate

Il **typedef Categoria** ha nidificato dentro un altro typedef come **proprietà**, ovvero i prodotti.

Lo stamp di questi ultimi verrà eseguito da una query **resolver** dedicata (Category).

Nel momento della chiamata, lato **client**, verrà fornito l'id della categoria e verrà restituita quest'ultima con associati tutti i prodotti.



Nota: il parametro usanto in questa chiamata sarà il **parent** che passa il valore del padre (la categoria) al figlio (l'array di prodotti).

Figura 6 La variabile fornita lato client sarà il categoryId

2. **Relazione Figlio → Genitore**

typeDefs

```
type Products {
    id: ID!
    name: String
    description: String
    image: String
    quantity: Int
    price: Float
    onSale: Boolean
    category: Category
}
```

resolver

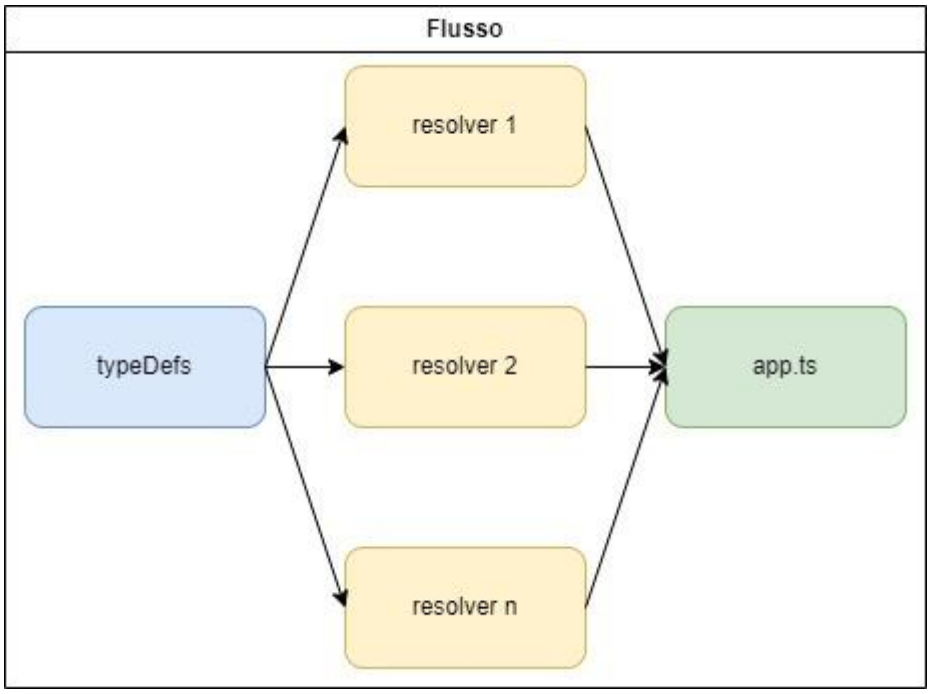
```
Products: {
    category: (parent: any, args: any) => {
        let idCategoria = parent.categoryId
        let arr = [];
        data.categories.find((categoria) => {
            if (categoria.id === idCategoria) {
                arr.push(categoria)
            }

        })
        return arr[0]
    }
}
```

client

```
query Products {
  products {
    name
    price
    quantity
    category {
      name
    }
  }
}
```

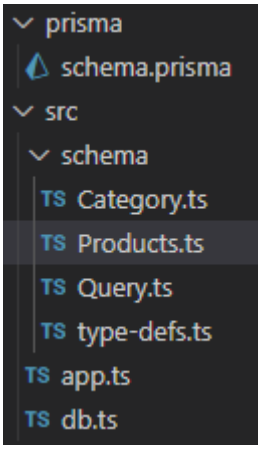
Figura 7 Flusso inverso che mostra la categoria a cui appartiene il prodotto

Cap.5 – Setup Directory



La configurazione di un progetto in graphQL richiede una divisione in **entry/output point** (app.ts o index.js), **schema**, costituito da typedefs e resolvers, questi ultimi suddivisi in sottofiles.

Esempio di **configurazione base**, con uso dell'ORM Prisma, di resolvers splittati in 3 files (e un finto db con dati fake per testing).




```
const server = new ApolloServer({
    typeDefs, resolvers: { Query, Category, Products },
    context: {
        bob: 1,
        sayHello: () => { console.log('Hi') }
    }
});
```

Figura 8 App.ts

Context: Attributo che permette di istanziare una variabile, funzione, ecc… per poi poterla passare in tutti i resolvers.

```
category: (parent: any, args: any, context: any) => {
    // console.log(context)
    const { sayHello } = context;
    sayHello();
```

Figura 9 Resolver Products

Nota: È possibile riordinare il progetto splittando ulteriormente i typedefs dello schema, ponendoli in files distinti, il tutto utilizzando un **workaround**.

Creazione di un file typeDefs con le rispettive types, query e mutation a sé stante

```
export const typesProducts = `
    type Products {
        id: ID!
        name: String
        description: String
        image: String
        quantity: Int
        price: Float
        onSale: Boolean
        category: Category
    }
`.replace(/\n/g, ' ') // il replace permette l'import senza \n

export const queriesProducts = `
    type Query {
        products: [Products]
        product(id: ID!): Products
    }
`.replace(/\n/g, ' ')

export const mutationProducts = `
`.replace(/\n/g, ' ')
```

Figura 10 Uso del regex per importare sottoforma di stringa prva di \n

Import nel file indice typedefs dei singoli typedefs

```
import { typesProducts, queriesProducts, mutationProducts } from './Products-Defs'
```

Inserimento del typeDefs specifico dei prodotti nell'instanziamento del typedefs

```
const typeDefs = gql`

${typesProducts}

    type Utente {
        id: ID!
        name: String
        username: String
    }
```

ATTENZIONE: Si tratta di un workaround in lavorazione!

Cap.6 – Query Filtering

- **Creazione di un filtro**

Es. Creazione di un filtro che restituisce una lista di prodotti che sono in vendita (**onSale: true**).

È possibile integrare filtri che permettono di restituire array di oggetti con determinate caratteristiche

Integriamo nei nostri typeDefs un input di filtro nella chiamata che, i precedenza, restituiva soltanto tutti i prodotti

```
type Query {
    products(filter: ProductsFilterInput): [Products]
```

Per essere costruiamo un input che considera soltanto la **proprietà onSale.**

Nota: è possibile costruire il filtro della complessità voluta, basta definire un **typeDefs ad hoc** (ricorda di non usare lo **stesso typeDefs** dei prodotti, il **nonullable** bloccherebbe i parametri non richiesti.)

```
input ProductsFilterInput {
    onSale: Boolean
}
```

```
const Query = {
    products: (parent, args) => {
        console.log(args.filter);
        let filter = args.filter
        let filteredProducts = db.products
        let products = []
        if (filter) {
            if (filter.onSale === true) {
                filteredProducts.map((product) => {
                    if (product.onSale === true) {
                        products.push(product)
                    }
                })
                return products
            } else {
                filteredProducts.map((product) => {
                    if (product.onSale === false) {
                        products.push(product)
                    }
                })
                return products
            }
        } else {
            return db.products
        }
    },
```

Il parametro args contiene il filtro che a sua volta ha le proprità definibile per filtrare la lista di elementi voluti.

Nota: il presente filtraggio può essere gestito in modi migliori.

Figura 11 Risultato della query dei prodotti adattata per gestire il filtro

Lato **client** basterà effettuare la query aggiungendo (se ve ne è il bisogno) il parametro con il valore voluto del filtro.

```
query {
  products(filter: {onSale: false}) {
    name
    price
    name
  }
}
```

Nota: **è possibile aggiungere il filtro anche dentro una lista nidificata!**

Es. Stampiamo una categoria e tutti i prodotti appartenenti ad essa

```
query Cat($categoryId: ID!) {
  category(id: $categoryId) {
    name
    products(filter: {onSale: false}) {
      name
      price
      onSale
    }
  }
}
```

Qui abbiamo la query **Cat** che è filtrata per un **id** che restituisce i prodotti, a loro volta **filtrati** per **onSale = false**.

Categoria con parametri filtrabile.

```
type Category {
    id:ID!
    name: String!
    products(filter: ProductsFilterInput): [Products]
}
```

```
const Category = {
    products: (parent: any, args: any) => {
        let idCategoria = parent.id
        console.log(idCategoria);
        let arr = []
        db.products.find((product) => {
            if (product.categoryId === idCategoria) {
                arr.push(product)
            }
        })
        let filter = args.filter
        let filteredProducts = arr
        let products = []
        if (filter) {
            if (filter.onSale === true) {
                filteredProducts.map((product) => {
                    if (product.onSale === true) {
                        products.push(product)
```

Resolver con nidificata la chiamata ai prodotti filtrabili a loro volta.

```
if (filter) {
    const { onSale, avgRating } = filter;
    // Filtro onSale
    if (onSale === true) {···
    } else {···
    }

    // Filtro avgRating
    if (avgRating) {

    }
```

Esempio di filtro composto da più proprietà lato resolver.

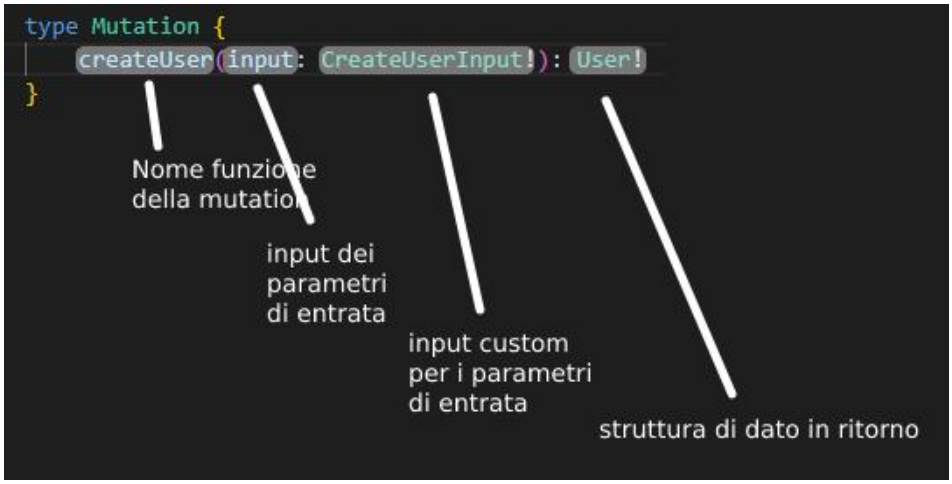
Le prop. In questione sono onSale e avgRating.

Cap.5 - Mutation

Tipologia di query con lo scopo di **modificare**, **cancellare** o **aggiungere** una risorsa.

Al pari delle post, put e delete permettono di far mutare lo stato dei dati, modificandoli.

- **POST**



I tipi di Mutation non si differenziano in base alle classiche distinzioni delle API Rest, ma la loro effettività cambia in base ai dati che vengono loro inviate.

Nell'esempio viene creato (nel **typeDefs**) un utente utilizzando come parametri di entrata

Figura 12 Struttura di una mutation atta a creare un utente

l'oggetto (type) **CreateUserInput**,

creato appositamente per non usare direttamente il type User, che potrebbe avere dei nullable che porterebbero a dei conflitti.

Es. Aggiunta di una categoria

```
input addCategoryInput {
    name: String!
}

type Mutation {
    addCategory(input: addCategoryInput): Category
}
```

Creiamo il type **Mutation** ed inseriamo dentro la mutation atta ad aggiungere un item.

Usiamo come struttura dati dell'input un type fatto appositamente (**addCategoryInput**).

Costruiamo un altro file dove inserire il **resolver** per la nuova mutation.

(In questo caso viene unicamente inserito un nuovo item in un **fake array**).

```
const Mutation = {
    addCategory: (parent: any, args: any) => {
        const newCategory = {
            id: "",
            name: args.input.name
        }
        db.categories.push(newCategory)
        return newCategory
    }
}

export { Mutation };
```

- **DELETE**

L'eliminazione dei dati è molto simile all'updating o alla chiamata di un item con uno specifico id, richiedendo, per l'appunto, solo un id per identificare una risorsa ed eliminarla da un db.

```
mutation deleteCategory {
  deleteCategory(id: "c2af9adc-d0b8-4d44-871f-cef66f86f7f6")
}
```

Figura 13 Mutation delete per eliminare una categoria

2 appunti:

- Considera sempre se l'item da eliminare abbia legami con altri e come rapportarti di conseguenza (che fine faranno i prodotti con quella categoria in eliminazione?)
- Nell'eliminazione, ciò che verrà restituito dalla mutation potrà essere uno scalar, come ad esempio un Boolean, che indicherà la corretta, o meno, eliminazione dell'item.

```
type Mutation {
    deleteCategory(id: ID!): Boolean
```

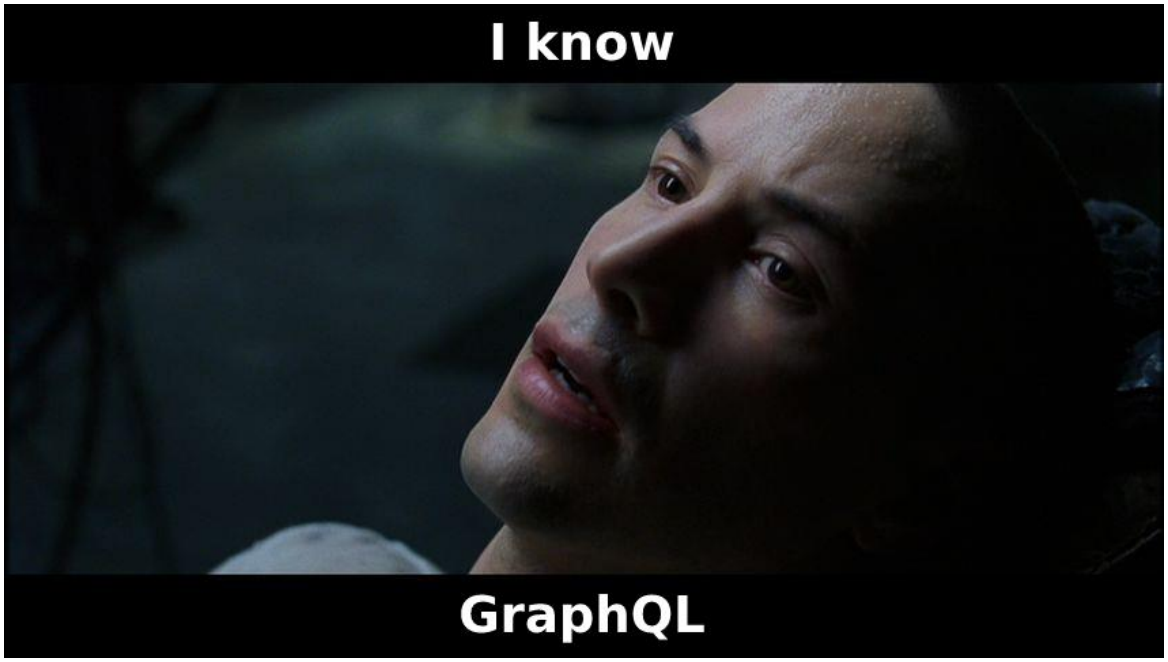
Figura 14 Vero o falso?

- **PUT**

L'updating dei dati usa dei meccanismi simili al delete e al post, utilizzando un id atto a identificare la risorsa da modificare e un input che struttura i dati da modificare.

```
mutation updatedCategory {
  updateCategory(id: "d914aec0-25b2-4103-9ed8-225d39018d1d",
  input: {
    name: "Kitchen and Bath"
  }
  ) {
    id
    name
  }
}
```

Figura 15 Esempio di update lato client di una categoria



***ATTENZIONE**: Le informazioni contenute in queste dispense potrebbero contenere errori o imprecisioni!