GRAPHQL

Questa è una miniguida per l'utilizzo di GraphQL (da https://graphql.org/learn)

Introduzione

GraphQL è un linguaggio che utilizza query per le API, eseguendo runtime lato server le query utilizzando un sistema di tipi definito per i dati. GraphQL non è legato ad alcun database o motore di archiviazione specifico ed è invece supportato dal codice e dai dati.

Un servizio GraphQL viene creato definendo tipi e campi dei tipi, quindi fornendo funzioni per ciascun campo e tipo.
Ad esempio, un servizio GraphQL che ti dice chi è l'utente che ha effettuato l'accesso e il nome di quell'utente potrebbe assomigliare a questo:

```
type Query {
  me: User
}

type User {
  id: ID
  name: String
}
```

Insieme alle funzioni per ciascun campo su ciascun tipo:

```
function Query_me(request) {
  return request.auth.user
}

function User_name(user) {
  return user.getName()
}
```

Dopo che un servizio GraphQL è in esecuzione (in genere presso un URL su un servizio Web), può ricevere query GraphQL da convalidare ed eseguire. Il servizio controlla innanzitutto una query per garantire che faccia riferimento solo ai tipi e ai campi definiti, quindi esegue le funzioni fornite per produrre un risultato.

Ad esempio, si arriverà ad un JSON:

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

Campi

Nella sua forma più semplice, GraphQL consiste nel richiedere campi specifici sugli oggetti.
Le query GraphQL possono utilizzare tutti i campi degli oggetti voluti, consentendo ai client di recuperare molti dati correlati in un'unica richiesta, invece di effettuare diversi viaggi di andata e ritorno come sarebbe necessario in una classica architettura REST.

```
{
  hero {
    name
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

```
{
  hero {
    name
    # Queries can have comments!
    friends {
      name
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
```

Argomenti

È possibile anche aggiungere argomenti ai campi. In GraphQL, ogni campo e oggetto nidificato può ottenere il proprio set di argomenti. Puoi anche passare argomenti in campi scalari, per implementare le trasformazioni dei dati una volta sul server, invece che su ogni client separatamente.
Gli argomenti possono essere di molti tipi diversi (es. si può utilizzare un enumerazione come unità di misura e utilizzare FOOT).
GraphQL viene fornito con un set predefinito di tipi, ma un server GraphQL può anche dichiarare i propri tipi personalizzati, purché possano essere serializzati nel formato di trasporto.

([https://graphql.org/learn/schema/)](https://graphql.org/learn/schema/)

```
{
  human(id: "1000") {
    name
    height
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 1.72
    }
  }
}
```

```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 5.6430448
    }
  }
}
```

Alias

Gli Alias, ti consentono di rinominare il risultato di un campo come preferisci.
(Nell'esempio sotto si possono vedere i due alias per il tipo hero)

```
{
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
```

```
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```

Fragments

GraphQL utilizza i fragements, cioè unità riutilizzabili, che ti consentono di costruire serie di cambi e includerli nelle query dove necessario.
Quindi viene utilizzato per suddividere complessi requisiti di dati dell'applicazione in blocchi più piccoli.

```
{
  leftComparison: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  rightComparison: hero(episode: JEDI) {
    ...comparisonFields
  }
}

fragment comparisonFields on Character {
  name
  appearsIn
  friends {
    name
  }
}
```

```
{
  "data": {
    "leftComparison": {
      "name": "Luke Skywalker",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "friends": [
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        },
        {
          "name": "C-3PO"
        },
```

E' anche possibile utilizzare delle variabili all'interno di questi fragments (Vedi Variabili più avanti)

```
query HeroComparison($first: Int = 3) {
  leftComparison: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  rightComparison: hero(episode: JEDI) {
    ...comparisonFields
  }
}

fragment comparisonFields on Character {
  name
  friendsConnection(first: $first) {
    totalCount
    edges {
      node {
        name
      }
    }
  }
}
```

VARIABLES

```
{
  "data": {
    "leftComparison": {
      "name": "Luke Skywalker",
      "friendsConnection": {
        "totalCount": 4,
        "edges": [
          {
            "node": {
              "name": "Han Solo"
            }
          },
          {
            "node": {
              "name": "Leia Organa"
            }
          },
          {
            "node": {
              "name": "C-3PO"
            }
          }
        ]
      }
    },
```

Nomi delle operazioni

La parola chiave della query che il nome della query. Negli esempi precedenti è sempre stata utilizzata una sintassi abbreviata dove non è necessaria. È molto utile nei documenti con più operazioni, o comunque molto complicati, però il suo uso è fortemente consigliato perché aiuta molto il debug e la registrazione lato server (es. quando qualcosa non va, migliori log).

Variabili

Molto spesso gli argomenti all'interno della query saranno dinamici (non schiantati come gli esempi precedenti).

Non sarebbe una buona idea passare questi argomenti dinamici direttamente nella stringa di query, perché in tal caso il nostro codice lato client dovrebbe manipolare dinamicamente la stringa di query in fase di runtime e serializzarla in un formato specifico di GraphQL. Invece, GraphQL offre un modo eccellente per fattorizzare i valori dinamici dalla query e passarli come un dizionario separato. Questi valori sono chiamati variabili.

Quando iniziamo a lavorare con le variabili, dobbiamo fare tre cose:

- Sostituiamo il valore statico nella query con $variableName
- Dichiariamo $variableName come una delle variabili accettate dalla query
- Passiamo variableName: valore nel dizionario delle variabili separato, specifico del trasporto (solitamente JSON).

```
query HeroNameAndFriends($episode: Episode)
  hero(episode: $episode) {
    name
    friends {
      name
    }
  }
}
```

VARIABLES

```
{
  "episode": "JEDI"
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
```

Ora, nel nostro codice client, possiamo semplicemente passare una variabile diversa invece di dover costruire una query completamente nuova. Questa è anche in generale una buona pratica per indicare quali argomenti nella nostra query dovrebbero essere dinamici: non dovremmo mai eseguire l'interpolazione di stringhe per costruire query da valori forniti dall'utente.

Definizione variabile

Le definizioni delle variabili sono la parte ($episode: Episode) nella query precedente. Funziona proprio come le definizioni degli argomenti per una funzione in un linguaggio digitato. Elenca tutte le variabili, precedute da $, seguite dal loro tipo, in questo caso Episode (tipo).
Le definizioni delle variabili possono essere facoltative o obbligatorie**, nel caso predente di Episode non abbiamo ! accanto al tipo, quindi è facoltativo**. Ma se il campo in cui stai passando la variabile richiede un argomento non nullo, anche la variabile deve essere obbligatoria.

Variabili predefinite

È inoltre possibile assegnare valori predefiniti alle variabili nella query aggiungendo il valore predefinito dopo la dichiarazione del tipo.

```
query HeroNameAndFriends($episode: Episode = JEDI) {
  hero(episode: $episode) {
    name
    friends {
      name
    }
  }
}
```

Quando vengono forniti valori predefiniti per tutte le variabili, è possibile chiamare la query senza passare alcuna variabile. Se delle variabili vengono passate come parte del dizionario delle variabili, sovrascriveranno le impostazioni predefinite.

Direttive

Potremmo aver bisogno di un modo per modificare dinamicamente la struttura e la forma delle nostre query utilizzando le variabili. Ad esempio, possiamo immaginare un componente dell'interfaccia utente che abbia una vista riepilogativa e dettagliata, in cui uno include più campi dell'altro.

```
query Hero($episode: Episode, $withFriends:
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}
```

VARIABLES

```
{
  "episode": "JEDI",
  "withFriends": false
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

Una direttiva può essere allegata a un campo o all'inclusione di un frammento e può influenzare l'esecuzione della query in qualsiasi modo desiderato dal server. La specifica GraphQL principale include esattamente due direttive, che devono essere supportate da qualsiasi implementazione del server GraphQL conforme alle specifiche:

- @include(if: Boolean) Include questo campo nel risultato solo se l'argomento è vero.
- @skip(if: Boolean) Salta questo campo se l'argomento è vero.

Le direttive possono essere utili per uscire da situazioni in cui altrimenti sarebbe necessario eseguire la manipolazione delle stringhe per aggiungere e rimuovere campi nella query. Le implementazioni del server possono anche aggiungere funzionalità sperimentali definendo direttive completamente nuove.

Mutazioni

La maggior parte delle discussioni su GraphQL si concentra sul recupero dei dati, ma qualsiasi piattaforma dati completa necessita di un modo per modificare anche i dati lato server.
In REST, qualsiasi richiesta potrebbe causare alcuni effetti collaterali sul server, ma per convenzione si suggerisce di non utilizzare richieste GET per modificare i dati. GraphQL è simile: tecnicamente qualsiasi

query potrebbe essere implementata per causare una scrittura di dati. Tuttavia, è utile stabilire una convenzione secondo cui qualsiasi operazione che causa scritture debba essere inviata esplicitamente tramite una mutazione.
Proprio come nelle query, se il campo di mutazione restituisce un tipo di oggetto, puoi richiedere campi nidificati. Questo può essere utile per recuperare il nuovo stato di un oggetto dopo un aggiornamento.
Ecco semplice esempio di mutazione:

```
mutation CreateReviewForEpisode($ep: Episod
  createReview(episode: $ep, review: $revie
    stars
    commentary
  }
}
```

VARIABLES

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!
    }
  }
}
```

Nota come il campo createReview restituisce le stelle e i campi dei commenti della recensione appena creata. Ciò è particolarmente utile quando si modificano i dati esistenti, ad esempio, quando si incrementa un campo, poiché possiamo modificare ed interrogare il nuovo valore del campo con una sola richiesta.
Si può anche notare che, in questo esempio, la variabile review che viene passata, non è uno scalare. È un tipo di oggetto di input, speciale, che può essere passato come argomento.

Campi multipli nelle mutazioni

Una mutazione può contenere più campi, proprio come una query. Esiste un'importante distinzione tra query e mutazioni, oltre al nome:

Mentre i campi di query vengono eseguiti in parallelo, i campi di mutazione vengono eseguiti in serie, uno dopo l'altro.

Ciò significa che se inviamo due mutazioni incrementaCredits in una richiesta, è garantito che la prima finirà prima dell'inizio della seconda, assicurandoci di non ritrovarci con una condizione di competizione con noi stessi.

Fragments in linea

Come molti altri sistemi di tipi, gli schemi GraphQL includono la possibilità di definire interfacce e tipi di unione. Se si sta interrogando un campo che restituisce un'interfaccia o un tipo di unione, si dovrà utilizzare i fragments in linea per accedere ai dati sul tipo concreto sottostante.
Ecco un esempio:

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
    ... on Human {
      height
    }
  }
}
```

VARIABLES

```
{
  "ep": "JEDI"
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "primaryFunction": "Astromech"
    }
  }
}
```

In questa query, il campo eroe restituisce il tipo Personaggio, che potrebbe essere un Umano o un Droide a seconda dell'argomento dell'episodio. Nella selezione diretta, puoi richiedere solo i campi esistenti nell'interfaccia Carattere, come il nome.

Per richiedere un campo sul tipo concreto, è necessario utilizzare un fragment in linea con una condizione di tipo. Poiché il primo fragment è etichettato come ... on Droid, il campo primaryFunction verrà eseguito solo se il personaggio restituito da hero è di tipo Droid. Allo stesso modo per il campo dell'altezza per il tipo Umano.

Anche i fragments con nome possono essere utilizzati allo stesso modo, poiché a un frammento con nome è sempre allegato un tipo.

Meta field

Dato che ci sono alcune situazioni in cui non sai quale tipo riceverai dal servizio GraphQL, hai bisogno di un modo per determinare come gestire tali dati sul client. GraphQL ti consente di richiedere __typename, un meta campo, in qualsiasi punto di una query per ottenere il nome del tipo di oggetto in quel punto.

```
{
  search(text: "an") {
    __typename
    ... on Human {
      name
    }
    ... on Droid {
      name
    }
    ... on Starship {
      name
    }
  }
}
```

```
{
  "data": {
    "search": [
      {
        "__typename": "Human",
        "name": "Han Solo"
      },
      {
        "__typename": "Human",
        "name": "Leia Organa"
      },
      {
        "__typename": "Starship",
        "name": "TIE Advanced x1"
      }
    ]
```

Nella query precedente, la ricerca restituisce un tipo di unione che può essere una delle tre opzioni. Sarebbe impossibile distinguere i diversi tipi dal client senza il campo __typename.
I servizi GraphQL forniscono alcuni meta campi, il resto dei quali viene utilizzato per esporre il sistema Introspection.

La guida presa in carico è molto più completa, con ancora informazioni su:

- Schema e Tipi
- Validazioni
- Esecuzione
- Autorizzazione
- Paginazione
- Caching
- Best Practise
- Ecc…

Questa vuole essere solo una guida introduttiva e non completa.