

# **GRAPHQL**

**FORMATEUR : CHACHIA ABDELILAH**

**CONTACT: [achachia2003@yahoo.fr](mailto:achachia2003@yahoo.fr)**

## **SOMMAIRE**

**1. Introduction**

**2. GraphQL - Composants d'application**

**3. GraphQL - Système de types**

**4. Résolveur**

**5. Requête**

**6. Mutation**

**7. Validation**

# INTRODUCTION

GraphQL est une technologie côté serveur open source qui a été développée par Facebook pour optimiser les appels d'API RESTful. C'est un moteur d'exécution et un langage de requête de données. Dans ce chapitre, nous discutons des avantages de l'utilisation de GraphQL.

## Pourquoi GraphQL

Les API RESTful suivent une approche claire et bien structurée axée sur les ressources. Cependant, lorsque les données deviennent plus complexes, les itinéraires s'allongent. Parfois, il n'est pas possible de récupérer des données avec une seule requête. C'est là que GraphQL devient utile. GraphQL structure les données sous la forme d'un graphique avec sa puissante syntaxe de requête pour parcourir, récupérer et modifier les données.

## Voici les avantages de l'utilisation du langage de requête GraphQL

Envoyez une requête GraphQL à votre API et obtenez exactement ce dont vous avez besoin. Les requêtes GraphQL renvoient toujours des résultats prévisibles. Les applications utilisant GraphQL sont rapides et stables. Contrairement aux services Restful, ces applications peuvent restreindre les données qui doivent être extraites du serveur.

L'exemple suivant vous aidera à mieux comprendre cela -

Considérons un objet métier Student avec les attributs id, firstName, lastName et collegeName . Supposons qu'une application mobile n'ait besoin de récupérer que firstName et id . Si nous concevons un point de terminaison REST comme /api/v1/students , il finira par récupérer des données pour tous les champs d'un objet étudiant . Cela signifie que les données sont surexploitées par le service RESTful.

**Ce problème peut être résolu en utilisant GraphQL. Considérez la requête GraphQL donnée ci-dessous :**

```
{
  students {
    id
    firstName
  }
}
```

**Cela renverra des valeurs uniquement pour les champs id et firstname. La requête ne récupérera pas les valeurs des autres attributs de l'objet étudiant. La réponse de la requête illustrée ci-dessus est comme indiqué ci-dessous :**

```
{
  "data": {
    "students": [
      {
        "id": "S1001",
        "firstName": "Mohtashim"
      },
      {
        "id": "S1002",
        "firstName": "Kannan"
      }
    ]
  }
}
```

### **Obtenez de nombreuses ressources en une seule requête**

**Les requêtes GraphQL aident à récupérer en douceur les objets métier associés, tandis que les API REST typiques nécessitent un chargement à partir de plusieurs URL. Les API GraphQL récupèrent toutes les données dont votre application a besoin en une seule requête. Les applications utilisant GraphQL peuvent être rapides même sur des connexions de réseau mobile lentes.**

**Considérons un autre objet métier, College qui a les attributs : nom et emplacement. L' objet métier Student a une relation d'association avec l'objet College. Si nous devons utiliser une API REST pour récupérer les détails des étudiants et de leur collège, nous finirions par faire deux requêtes au serveur comme /api/v1/students et /api/v1/colleges . Cela entraînera une sous-extraction des données à chaque demande. Les applications mobiles sont donc obligées d'effectuer plusieurs appels au serveur pour obtenir les données souhaitées.**

**Cependant, l'application mobile peut récupérer les détails des objets Étudiant et Collège en une seule requête en utilisant GraphQL.**

**Ce qui suit est une requête GraphQL pour récupérer des données :**

```
{
  students{
    id
    firstName
    lastName
    college{
      name
      location
    }
  }
}
```

**La sortie de la requête ci-dessus contient exactement les champs que nous avons demandés, comme indiqué ci-dessous -**

```

{
  "data": {
    "students": [
      {
        "id": "S1001",
        "firstName": "Mohtashim",
        "lastName": "Mohammad",
        "college": {
          "name": "CUSAT",
          "location": "Kerala"
        }
      },
      {
        "id": "S1002",
        "firstName": "Kannan",
        "lastName": "Sudhakaran",
        "college": {
          "name": "AMU",
          "location": "Uttar Pradesh"
        }
      },
      {
        "id": "S1003",
        "firstName": "Kiran",
        "lastName": "Panigrahi",
        "college": {
          "name": "AMU",
          "location": "Uttar Pradesh"
        }
      }
    ]
  }
}

```

## Décrire ce qui est possible avec un système de type

GraphQL est fortement typé et les requêtes sont basées sur les champs et leurs types de données associés. En cas d'incompatibilité de type dans une requête GraphQL, les applications serveur renvoient des messages d'erreur clairs et utiles. Cela facilite le débogage et la détection facile des bogues par les applications clientes. GraphQL fournit également des bibliothèques côté client qui peuvent aider à réduire la conversion et l'analyse explicites des données.

Un exemple des types de données Étudiant et Collège est donné ci-dessous :

```
type Query {  
  students:[Student]  
}
```

```
type Student {  
  id:ID!  
  firstName:String  
  lastName:String  
  fullName:String  
  college:College  
}
```

```
type College {  
  id:ID!  
  name:String  
  location:String  
  rating:Float  
  students:[Student]  
}
```

# GraphQL - Composants d'application

Ce chapitre traite des différents composants GraphQL et de la manière dont ils communiquent entre eux. L'ensemble des composants de l'application peut être distingué comme ci-dessous :

1. Composants côté serveur
2. Composants côté client

## Composants côté serveur

Le serveur GraphQL constitue le composant central côté serveur et permet d'analyser les requêtes provenant des applications clientes GraphQL. Apollo Server est l'implémentation la plus couramment utilisée de la spécification GraphQL. Les autres composants de programmation de serveur incluent les éléments suivants :

### Schéma

Un schéma GraphQL est au centre de toute implémentation de serveur GraphQL et décrit les fonctionnalités disponibles pour les clients qui s'y connectent.

### Requete

Une requête GraphQL est la requête de l'application cliente pour récupérer des données à partir d'une base de données ou d'API héritées.

### Résolveur

Les résolveurs fournissent les instructions pour transformer une opération GraphQL en données. Ils résolvent la requête en données en définissant des fonctions de résolution.

## Composants côté client

### GraphiQL

Interface basée sur un navigateur pour l'édition et le test des requêtes et des mutations GraphQL.

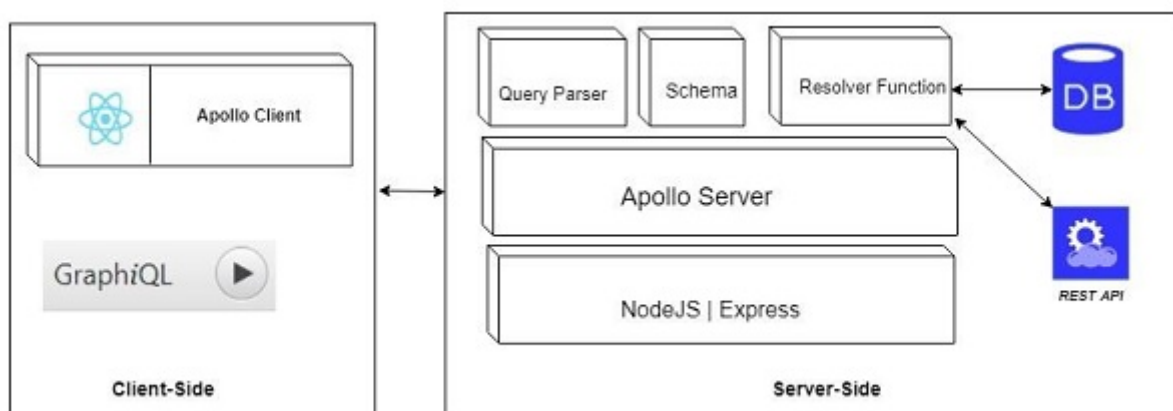
### ApolloClient

Meilleur outil pour créer des applications clientes GraphQL. S'intègre bien avec tous les front-end javascript.

Le schéma ci-dessous montre une architecture Client-Serveur . Le serveur Web est construit sur le framework NodeJs et Express. Une demande est faite au serveur Apollo GraphQL par l'application ReactJS (construite à l'aide de la bibliothèque Apollo Client) ou l'application de navigateur GraphiQL.

La requête sera analysée et validée par rapport à un schéma défini dans le serveur. Si le schéma de requête passe la validation, les fonctions de résolution associées seront exécutées.

Le résolveur contiendra du code pour récupérer les données d'une API ou d'une base de données.



**Code source :Exemple projet serveur GraphQL en Node.js**

<https://github.com/achachia/GraphQL-Next-U/tree/ecc67c8397f0238f4b33b9bdefa0620f5036e562>



# GraphQL - Système de types

GraphQL est un langage fortement typé. Type System définit divers types de données pouvant être utilisés dans une application GraphQL. Le système de type aide à définir le schéma, qui est un contrat entre le client et le serveur. Les types de données GraphQL couramment utilisés sont les suivants :

## Scalaire (Scalar )

Stocke une seule valeur

## Objet (Object)

Montre quel type d'objet peut être récupéré

## Requete(Query)

Type de point d'entrée vers d'autres types spécifiques

## Mutation(Mutation)

Point d'entrée pour la manipulation des données

## Énumération(Enum)

Utile dans une situation où vous avez besoin que l'utilisateur choisisse parmi une liste prescrite d'options

## Scalaire (Scalar )

Les types scalaires sont des types de données primitifs qui ne peuvent stocker qu'une seule valeur. Les types scalaires par défaut proposés par GraphQL sont :

1. **Int** :Entier 32 bits signé
2. **Float** :Valeur à virgule flottante double précision signée
3. **String** : UTF - Séquence de 8 caractères
4. **Boolean** : Vrai ou faux
5. **ID** : Un identifiant unique, souvent utilisé comme identifiant unique pour récupérer un objet ou comme clé pour un cache.

La syntaxe pour définir un type scalaire est la suivante -

**field: data\_type**

L'extrait de code ci-dessous définit un champ nommé salutation qui renvoie une valeur de chaîne.

**greeting: String**

## Type d'objet

Le type d'objet est le type le plus couramment utilisé dans un schéma et représente un groupe de champs. Chaque champ à l'intérieur d'un type d'objet correspond à un autre type, autorisant ainsi les types imbriqués.

En d'autres termes, un type d'objet est composé de plusieurs types scalaires ou types d'objets.

**La syntaxe pour définir un type d'objet est donnée ci-dessous :**

```
type object_type_name  
{  
  field1: data_type  
  field2:data_type  
  ....  
  fieldn:data_type  
}
```

**Vous pouvez considérer l'extrait de code suivant :**

**--Define an object type--**

```
type Student {  
  stud_id:ID  
  firstname: String  
  age: Int  
  score:Float  
}
```

**--Defining a GraphQL schema--**

```
type Query  
{  
  stud_details:[Student]  
}
```

**L'exemple donné ci-dessus définit un objet de type de données Student. Le champ stud\_details dans le schéma racine Query renverra une liste d'objets Student.**

## Type de requête

Une requête GraphQL est utilisée pour récupérer des données. C'est comme demander une ressource dans les API basées sur REST. Pour faire simple, le type de requête est la requête envoyée par une application cliente au serveur GraphQL. GraphQL utilise le langage de définition de schéma (SDL) pour définir une requête. Le type de requête est l'un des nombreux types de niveau racine dans GraphQL.

La syntaxe pour définir une requête est comme indiqué ci-dessous :

```
type Query {  
  field1: data_type  
  field2: data_type  
  
  field3(param1: data_type, param2: data_type, ...paramN: data_type): data_type  
}
```

Un exemple de définition d'une requête :

```
type Query {  
  greeting: String  
}
```

## Type de mutation

Les mutations sont des opérations envoyées au serveur pour créer, mettre à jour ou supprimer des données. Ceux-ci sont analogues aux verbes PUT, POST, PATCH et DELETE pour appeler les API basées sur REST.

La mutation est l'un des types de données de niveau racine dans GraphQL. Le type Query définit les points d'entrée pour les opérations de récupération de données tandis que le type Mutation spécifie les points d'entrée pour les opérations de manipulation de données.

**La syntaxe pour définir un type de mutation est donnée ci-dessous :**

```
type Mutation {  
    field1: data_type  
  
    field2(param1:data_type,param2:data_type,...paramN:data_type):data  
_type  
}
```

**Par exemple, nous pouvons définir un type de mutation pour ajouter un nouvel étudiant comme ci-dessous :**

```
type Mutation {  
    addStudent(firstName: String, lastName: String): Student  
}
```

## **Type d'énumération**

**Un Enum est similaire à un type scalaire. Les énumérations sont utiles dans une situation où la valeur d'un champ doit provenir d'une liste d'options prescrite.**

**La syntaxe pour définir un type Enum est -**

```
type enum_name{  
    value1  
    value2  
}
```

**L'extrait suivant illustre comment un type enum peut être défini :**

```
type Days_of_Week{  
    SUNDAY  
    MONDAY  
    TUESDAY  
    WEDNESDAY  
    THURSDAY  
    FRIDAY  
    SATURDAY  
}
```

## **Type de liste**

**Les listes peuvent être utilisées pour représenter un tableau de valeurs d'un type spécifique. Les listes sont définies avec un modificateur de type [] qui enveloppe les types d'objets, les scalaires et les énumérations.**

**La syntaxe suivante peut être utilisée pour définir un type de liste :**

```
field:[data_type]
```

**L'exemple ci-dessous définit un type de liste todos :**

```
type Query {  
    todos: [String]  
}
```

## **Type non nullable**

**Par défaut, chacun des types scalaires principaux peut être défini sur null**

En d'autres termes, ces types peuvent soit renvoyer une valeur du type spécifié, soit ne pas avoir de valeur. Pour remplacer cette valeur par défaut et spécifier qu'un champ doit être défini, un point d'exclamation (!) peut être ajouté à un type. Cela garantit la présence de valeur dans les résultats renvoyés par la requête.

La syntaxe suivante peut être utilisée pour définir un champ non nullable :

**field:data\_type!**

Dans l'exemple ci-dessous, **stud\_id** est déclaré comme champ obligatoire.

```
type Student {  
  stud_id:ID!  
  firstName:String  
  lastName:String  
  fullName:String  
  college:College  
}
```

## Résolveur

Resolver est une collection de fonctions qui génèrent une réponse pour une requête GraphQL. En termes simples, un résolveur agit comme un gestionnaire de requêtes GraphQL. Chaque fonction de résolution dans un schéma GraphQL accepte quatre arguments positionnels comme indiqué ci-dessous :

`fieldName:(root, args, context, info) => { result }`

Un exemple de fonctions de résolveur est présenté ci-dessous :

```
//resolver function with no parameters and returning string
greeting:() => {
  return "hello from  Tutorialspoint !!!"
}
```

```
//resolver function with no parameters and returning list
students:() => db.students.list()
```

```
//resolver function with arguments and returning object
studentById:(root,args,context,info) => {
  return db.students.get(args.id);
}
```

Ci-dessous sont les arguments positionnels et leur description :

### 1.root

L'objet qui contient le résultat renvoyé par le résolveur sur le champ parent.



## 2. args

Un objet avec les arguments passés dans le champ de la requête.

## 3 . context

Il s'agit d'un objet partagé par tous les résolveurs d'une requête particulière.

## 4. info

Il contient des informations sur l'état d'exécution de la requête, notamment le nom du champ, le chemin d'accès au champ depuis la racine.

### Etape 1 : Créer un schéma

Ajoutez le fichier `schema.graphql` dans le dossier du projet et ajoutez le code suivant :

```
type Query {  
  greeting:String  
  students:[Student]  
  studentById(id:ID!):Student  
}
```

```
type Student {  
  id:ID!  
  firstName:String  
  lastName:String  
  password:String  
  collegeId:String  
}
```

## Étape 2 - Créer un résolveur

```
const db = require('./db');

const Query = {
  students:() => db.students.list(),
  college:() => db.colleges.list(),
  studentById:(root,args,context,info) => {
    //args will contain parameter passed in query
    return db.students.get(args.id);
  }
}

const Student = {
  fullName:(root,args,context,info) => {
    return root.firstName+":"+root.lastName
  },
  college:(root) => {
    return db.colleges.get(root.collegeId);
  }
}

const College = {
  studentsByCollege:(root) => {
    return db.students.list().filter(function(item) {
      return item.collegeId == root.id;
    });
  }
}

module.exports = {Query,Student, College}
```

Ici, `studentById` prend en compte trois paramètres. Comme discuté dans ce chapitre, le `studentId` peut être récupéré à partir de `args` ; `root` contiendra l'objet `Query` lui-même. Pour renvoyer un étudiant spécifique, nous devons appeler la méthode `get` avec le paramètre `id` dans la collection des étudiants.

Ici, `salutation`, `étudiants`, `studentById` sont les résolveurs qui gèrent la requête. La fonction de résolution des étudiants renvoie une liste d'étudiants de la couche d'accès aux données. Pour accéder aux fonctions de résolution en dehors du module, l'objet `Query` doit être exporté à l'aide de `module.exports`.

### Étape 3 - Exécutez l'application

Créez un fichier `server.js`. Reportez-vous à l'étape 8 du chapitre Configuration de l'environnement. Exécutez la commande `npm start` dans le terminal. Le serveur sera opérationnel sur le port 9000. Ici, nous utilisons `GraphiQL` comme client pour tester l'application.

Ouvrez le navigateur et entrez l'url, `http://localhost:9000/graphiql`. Tapez la requête suivante dans l'éditeur

```
{
  studentById(id:"S1001") {
    id
    firstName
    lastName
  }
}
```

La sortie de la requête ci-dessus est comme indiqué ci-dessous :

```
{  
  "data": {  
    "studentById": {  
      "id": "S1001",  
      "firstName": "Mohtashim",  
      "lastName": "Mohammad"  
    }  
  }  
}
```

# Requête

Une opération GraphQL peut être une opération de lecture ou d'écriture. Une requête GraphQL est utilisée pour lire ou récupérer des valeurs tandis qu'une mutation est utilisée pour écrire ou publier des valeurs. Dans les deux cas, l'opération est une simple chaîne qu'un serveur GraphQL peut analyser et répondre avec des données dans un format spécifique. Le format de réponse populaire généralement utilisé pour les applications mobiles et Web est JSON.

## Illustration 1 - Requête du modèle étudiant avec un champ personnalisé

Tapez la requête suivante dans l'éditeur :

```
{  
  students{  
    id  
    fullName  
  }  
}
```

La réponse à la requête est donnée ci-dessous :

```
{
  "data": {
    "students": [
      {
        "id": "S1001",
        "fullName": "Mohtashim:Mohammad"
      },

      {
        "id": "S1002",
        "fullName": "Kannan:Sudhakaran"
      },

      {
        "id": "S1003",
        "fullName": "Kiran:Panigrahi"
      }
    ]
  }
}
```

## Illustration 2 - Requête imbriquée

**Tapez la requête suivante dans l'éditeur :**

```
{  
  students{  
    id  
    firstName  
    college {  
      id  
      name  
      location  
      rating  
    }  
  }  
}
```

**La réponse à la requête est donnée ci-dessous :**

```
{
  "data": {
    "students": [
      {
        "id": "S1001",
        "firstName": "Mohtashim",
        "college": {
          "id": "col-102",
          "name": "CUSAT",
          "location": "Kerala",
          "rating": 4.5
        }
      },
      {
        "id": "S1002",
        "firstName": "Kannan",
        "college": {
          "id": "col-101",
          "name": "AMU",
          "location": "Uttar Pradesh",
          "rating": 5
        }
      },
      {
        "id": "S1003",
        "firstName": "Kiran",
        "college": {
          "id": "col-101",
          "name": "AMU",
          "location": "Uttar Pradesh",
          "rating": 5
        }
      }
    ]
  }
}
```



# Mutation

Les requêtes de mutation modifient les données dans le magasin de données et renvoient une valeur. Il peut être utilisé pour insérer, mettre à jour ou supprimer des données. Les mutations sont définies comme faisant partie du schéma.


La syntaxe d'une requête de mutation est donnée ci-dessous -

```
mutation{  
  someEditOperation(dataField:"valueOfField"):returnType  
}
```

## Illustration :

Comprenons comment ajouter un nouvel enregistrement d'étudiant dans le magasin de données à l'aide d'une requête de mutation

🔑 02530ef788 ▾ GraphQL-Next-U / schema.graphql

 achachia Mutations-GraphQL-V1

🔍 1 contributor

28 lines (23 sloc) | 574 Bytes

```
1  type Query {  
2    students:[Student],  
3    studentById(id:ID!):Student,  
4    college : [College]  
5  }  
6  
7  type Student {  
8    id:ID!  
9    firstName:String  
10   lastName:String  
11   fullName:String  
12   college:College  
13 }  
14  
15 type College {  
16   id:ID!  
17   name:String  
18   location:String  
19   rating:Float  
20   studentsByCollege:[Student]  
21 }  
22  
23 type Mutation {  
24   returnStringByCreateStudent(collegeId:ID,firstName:String,lastName:String,email:String,password:String,age:Int):String,  
25   returnObjectByCreateStudent(collegeId:ID,firstName:String,lastName:String,email:String,password:String,age:Int):Student  
26 }  
27  
28
```



achachia Mutations-GraphQL-V1

1 contributor

51 lines (44 sloc) 1.26 KB

```
1  const db = require('./db');
2
3  const Query = {
4    students:() => db.students.list(),
5    college:() => db.colleges.list(),
6    studentById:(root,args,context,info) => {
7      //args will contain parameter passed in query
8      return db.students.get(args.id);
9    }
10 }
11
12 const Student = {
13   fullName:(root,args,context,info) => {
14     return root.firstName+" "+root.lastName
15   },
16   college:(root) => {
17     return db.colleges.get(root.collegeId);
18   }
19 }
20
21 const College = {
22   studentsByCollge:(root) => {
23     return db.students.list().filter(function(item) {
24       return item.collegeId == root.id;
25     });
26   }
27 }
28
29 const Mutation = {
30   returnStringByCreateStudent:(root,args,context,info) => {
31     return db.students.create({collegeId:args.collegeId,
32       firstName:args.firstName,
33       lastName:args.lastName,
34       email:args.email,
35       password:args.password,
36       age:args.age,})
37   },
38   returnObjectByCreateStudent:(root,args,context,info) => {
39     const id = db.students.create({collegeId:args.collegeId,
40       firstName:args.firstName,
41       lastName:args.lastName,
42       email:args.email,
43       password:args.password,
44       age:args.age,})
45     return db.students.get(id)
46   }
47 }
48
49
50
51 module.exports = {Query,Student, College,Mutation}
```

### Exemple requette mutation :

```
mutation {  
  returnObjectByCreateStudent(collegeId:"col-2",firstName:"Tim-  
    bis",lastName:"George-bis",email: "tim-  
    bis@gmail.com",password:"tim123-bis",age:25)  
  {  
    id  
    firstName  
    lastName  
  }  
}
```

### Requette équivalente :

```
mutation CreateStudent($input: StudentInput!) {  
  returnStringByCreateStudent(input: $input)  
}
```

### Query variables :

```
{"myname_Variable": "S1001", "input": {"collegeId":  
"col-101", "firstName": "hope is a good thing", "lastName": "hope is a  
good thing", "email": "toto@yopmail.com", "password": "hope is a  
good thing", "age": 25}}
```

### Code source Github (V1) :

[https://github.com/NEXT-U-WEBTECH/GraphQL-Next-U/  
tree/02530ef788d803e23d2de075534f188e89e5b4a7](https://github.com/NEXT-U-WEBTECH/GraphQL-Next-U/tree/02530ef788d803e23d2de075534f188e89e5b4a7)

## GraphQL - Validation

Lors de l'ajout ou de la modification de données, il est important de valider l'entrée de l'utilisateur. Par exemple, nous pouvons avoir besoin de nous assurer que la valeur d'un champ n'est toujours pas nulle. Nous pouvons utiliser ! (non-nullable) marqueur de type dans GraphQL pour effectuer une telle validation.

La syntaxe pour utiliser le ! le marqueur de type est comme indiqué ci-dessous :

```
type TypeName {  
  field1:String!,  
  field2:String!,  
  field3:Int!  
}
```

La syntaxe ci-dessus garantit que tous les champs ne sont pas nuls.

Si nous voulons implémenter des règles supplémentaires comme vérifier la longueur d'une chaîne ou vérifier si un nombre se trouve dans une plage donnée, nous pouvons définir des validateurs personnalisés.

La logique de validation personnalisée fera partie de la fonction de résolution.

Comprenons cela à l'aide d'un exemple.

### Illustration - Implémentation de validateurs personnalisés

```

33 1
34 v returnStringByCreateStudent:(root,args,context,info) => {
35
36     const {email,firstName,password} = args.input;
37
38     const emailExpression = /^((([^<>()\[\]\\\.,;:\s@"]+(\.[^<>()\[\]\\\.,;:\s@"]+)*)(("[.+]"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.([0-9]{1,3}\.([0-9]{1,3}\.([0-9]{1,3}\.([0-9]{1,3}\.([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$)/;
39
40     const isValidEmail = emailExpression.test(String(email).toLowerCase())
41
42 v     if(!isValidEmail){
43
44         throw new Error("Adresse mail non valide ")
45     }
46
47 v     if(firstName.length > 15){
48
49         throw new Error("firstName doit comporter moins de 15 caractères")
50     }
51
52 v     if(password.length < 8 ){
53         throw new Error("le mot de passe doit comporter au moins 8 caractères")
54     }
55
56 v     return db.students.create({collegeId:args.input.collegeId,
57     firstName:args.input.firstName,
58     lastName:args.input.lastName,
59     email:args.input.email,
60     password:args.input.password,
61     age:args.input.age,})
62
63 },

```

5aWagdW0[fZgT,

[https://github.com/NEXT-U-WEBTECH/GraphQL-Next-U/  
commit/23800aede6a4bc6ee08981f948d2f335ab4c59e5](https://github.com/NEXT-U-WEBTECH/GraphQL-Next-U/commit/23800aede6a4bc6ee08981f948d2f335ab4c59e5)