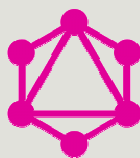


GraphQL



GraphQL

AYMEN SELLAOUTI
aymen.sellaouti@gmail.com

1

Références



GraphQL

AYMEN SELLAOUTI

2

Intro à GraphQL

AYMEN SELLAOUTI

3

Introduction à GraphQL



Objectifs

- Définir qu'est ce que GraphQL.
- Identifier les avantages de GraphQL.
- Présenter l'historique de GraphQL.

AYMEN SELLAOUTI

4

Introduction à GraphQL GraphQL un peu d'historique



GraphQL a été développé en 2012 par FB. Les raisons qui ont poussé FB à chercher une alternative aux API REST sont les suivantes :

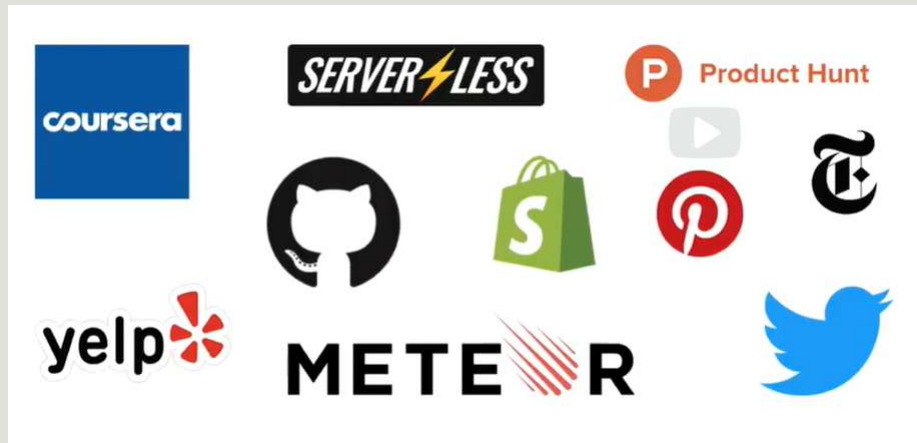
- La croissance de l'utilisation des mobiles ce qui implique un besoin d'optimisation du chargement des données.
- Explosion des Framework FrontEnd
- Le besoin d'accélérer le développement des API

Introduction à GraphQL GraphQL un peu d'historique



- GraphQL a été présenté au public en 2015 dans la conférence React.js.
- Il peut être utilisé par n'importe quel langage ou Framework
- D'autres entreprises tel que Coursera et Netflix (Falcor) ont entrepris des projets similaires qu'ils ont arrêté à la sortie de GraphQL et l'ont utilisé.

Introduction à GraphQL Ils ont choisi GraphQL



AYMENSELLAOUTI

7

Introduction à GraphQL Pourquoi GraphQL



- Over fetching
- Under fetching

- Plus rapide : On peut regrouper plusieurs requêtes en une seule
- Plus flexible : On peut demander ce qu'on veut pas besoin de spécifier coté serveur quoi retourner, c'est le client qui demande.

AYMENSELLAOUTI

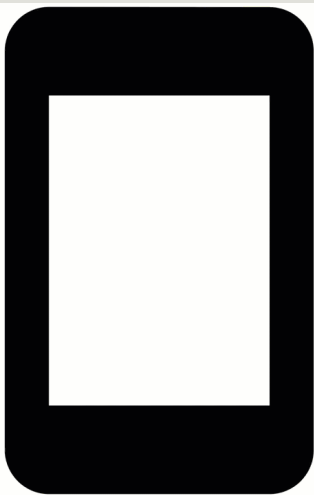
8

Pourquoi GraphQL

GraphQL Vs REST



Example: **Blogging App**



Pourquoi GraphQL

GraphQL Vs REST



REST	GraphQL
Multipl es requêtes	Une requête
Problème d’over et under fetching	Récupérer exactement ce que vous voulez
Il y a une dépendance entre les deux équipes frontend et backend.	Les deux équipes frontend et backend peuvent travailler séparément

Introduction à GraphQL Pourquoi GraphQL



- GraphQL crée des api **rapides** et **flexibles** permettant au client un **control complet** sur les données qu'il désire.
- GraphQL permet donc d'avoir **moins de requêtes HTTP** et moins de code à gérer.

GraphQL Big Picture



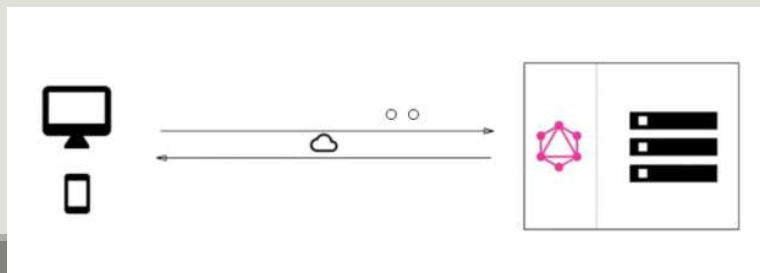
- GraphQL est une spécification. Il n'y a donc pas d'architecture particulière pour GraphQL.
- Cependant trois cas d'utilisations sont souvent rencontrés.
 - L'utilisation de GraphQL avec un nouveau projet d'une application connecté à une BD.
 - L'intégration de GraphQL avec un système existant
 - Une approche hybride entre les deux premiers

GraphQL Big Picture

GraphQL avec une application connecté à une BD



- Utilisé généralement avec de nouveau projets que vous entamer au départ.
- Généralement utilisée avec un seul serveur Web ou on expose l'ensemble des fonctionnalités.



AYMEN SELLAOUTI

13

GraphQL Big Picture

Intégrer GraphQL à un système existant



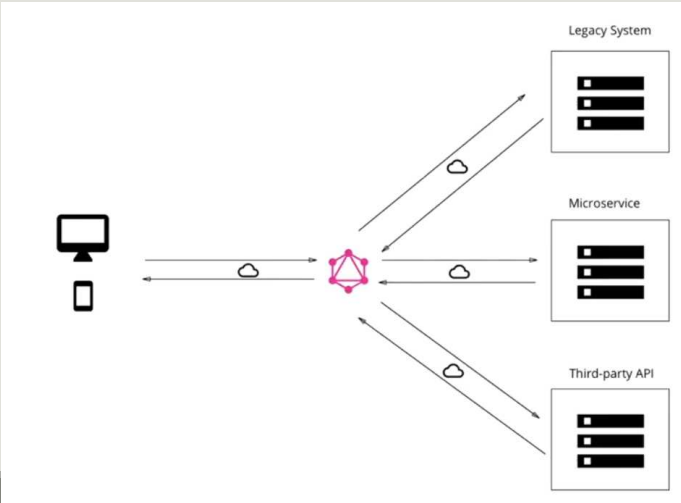
- Très utiles lorsque vous avez plusieurs API complexes. Plus votre application grandit plus vous avez de la documentation et de la complication
- GraphQL servira dans ce cas pour unifier le système existant en abstrayant toute la complexité derrière.
- Cette abstraction permettra au client de ne plus s'occuper de la source de données. Qu'elle soit via une BD ou un service web ou une API tierce tout est abstrait par GraphQL.

AYMEN SELLAOUTI

14

GraphQL Big Picture

Intégrer GraphQL à un système existant

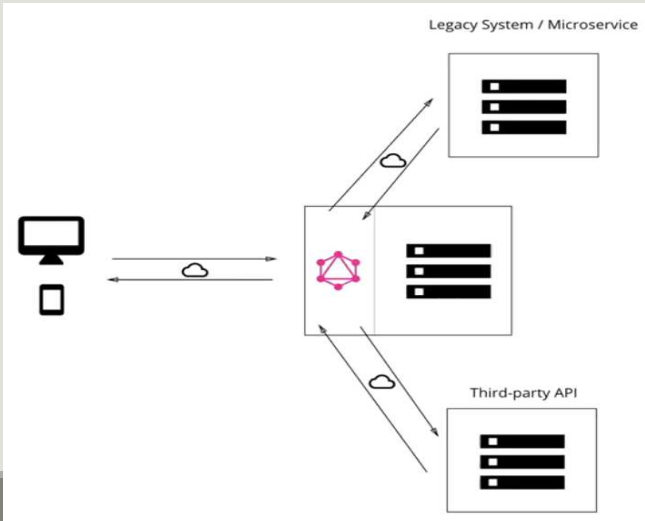


AYMENSELLAOUTI

15

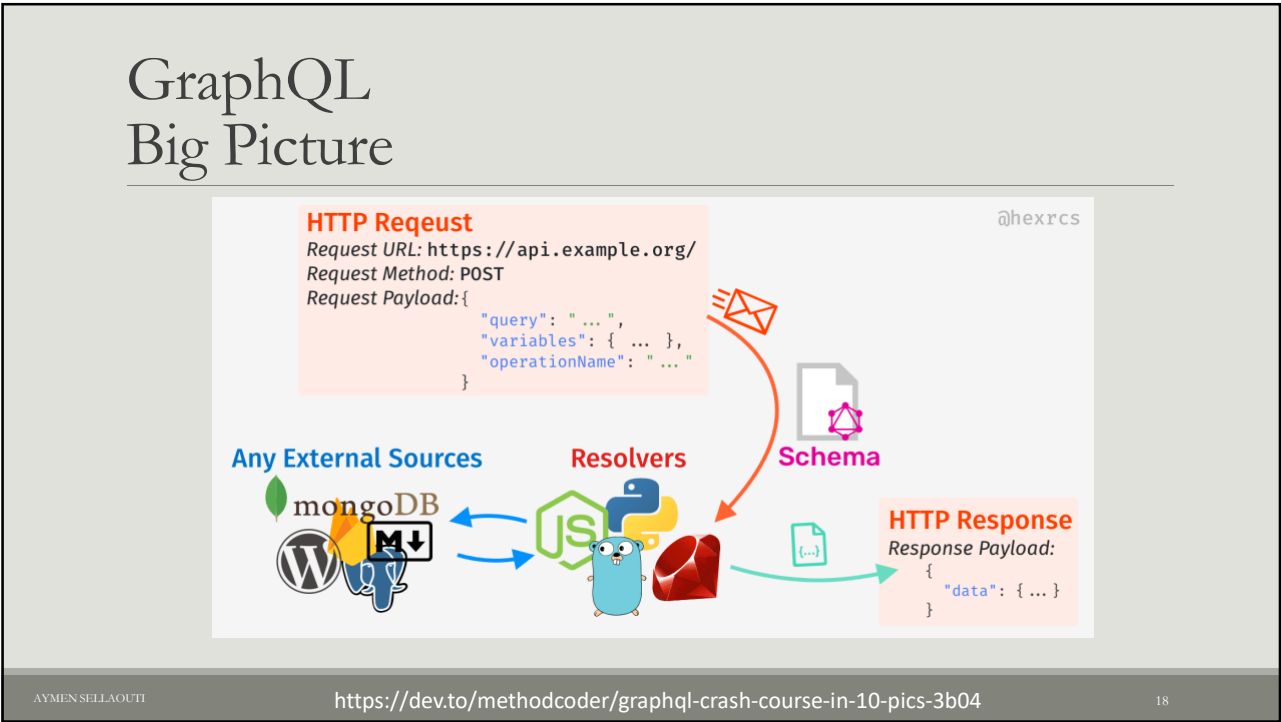
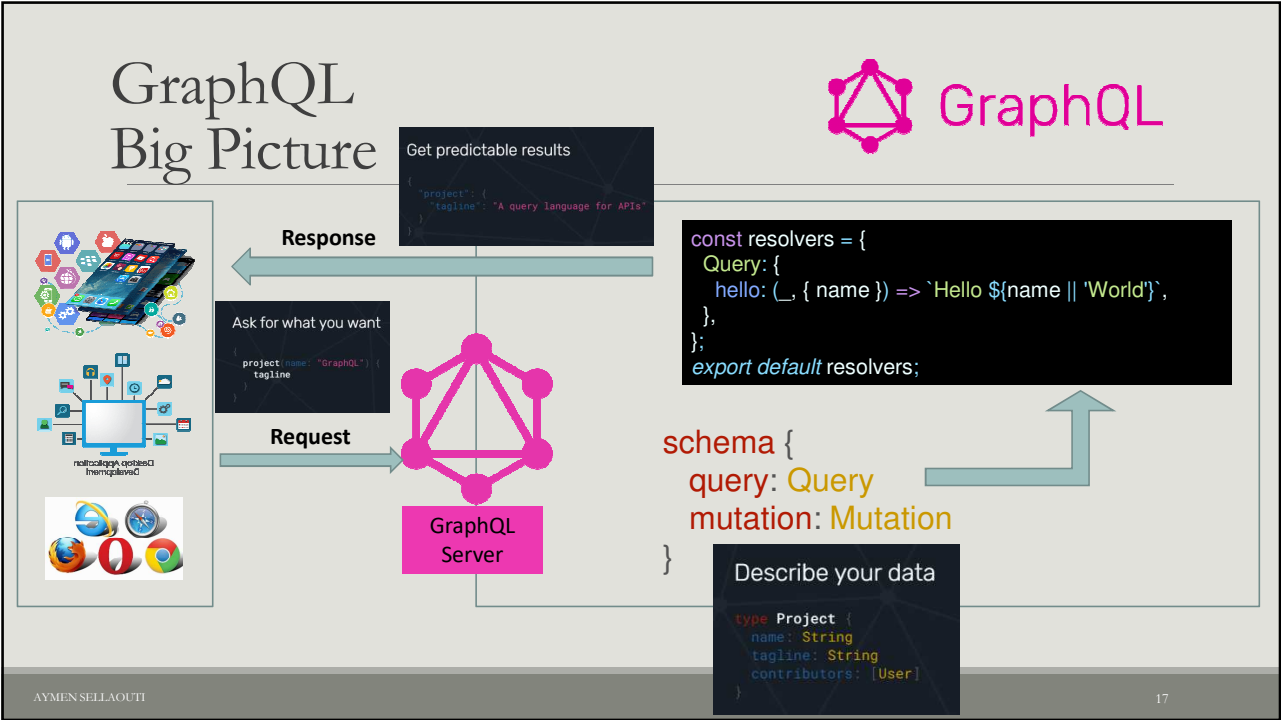
GraphQL Big Picture

Système hybride

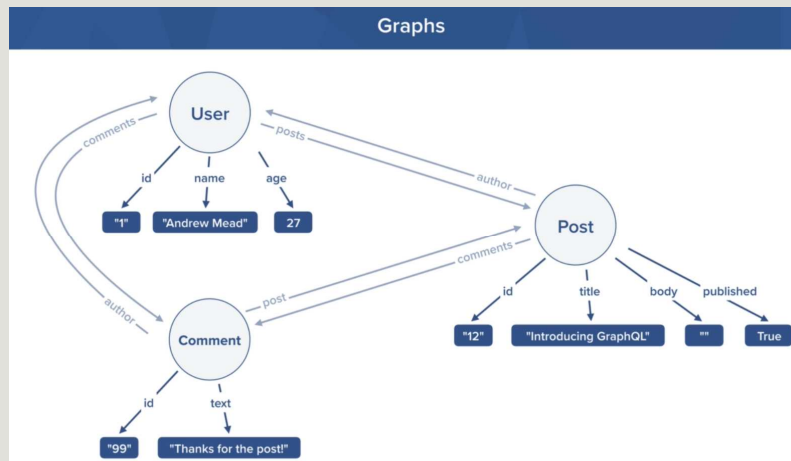


AYMENSELLAOUTI

16



GraphQL Big Picture



AYMEN SELLAOUTI

19

GraphQL Implémentation



- Etant une spécification, GraphQL est implémenté avec plusieurs langages.
- Nous allons utiliser JS et le serveur GraphQL Yoga



<https://github.com/prisma-labs/graphql-yoga>

AYMEN SELLAOUTI

<https://graphql.org/code/>

20



IMPLÉMENTER VOTRE SERVEUR GRAPHQL

AYMEN SELLAOUTI

21

GraphQL Implémentation



- Afin d'initialiser votre projet commencer par initialiser un projet node avec `npm init`.
- Installer le serveur graphql-yoga avec la commande :
`npm install graphql-yoga`
- Créer un fichier `index.js` et ajouter y le code suivant :

```
import { GraphQLServer } from 'graphql-yoga'
// ... or using "require()"
// const { GraphQLServer } = require('graphql-yoga')
const typeDefs = `
  type Query {
    hello(name: String): String!
  }
`;
const resolvers = {
  Query: {
    hello: (_, { name }) => `Hello ${name || 'World'}`,
  },
};
const server = new GraphQLServer({ typeDefs, resolvers })
server.start(() => console.log("Server is running on localhost:4000"))
```

AYMEN SELLAOUTI

<https://graphql.org/code/>

22

GraphQL Implémentation



- Afin de lancer le serveur GraphQL, vous devez créer une instance de la classe GraphQLServer.
- Son constructeur prend en paramètre plusieurs variables dont deux qui sont typeDefs et resolvers.
- typeDefs reçoit la définition du schéma. Ça peut être une chaîne de caractère ou un fichier « .graphql ».
- resolvers reçoit les resolvers.

```
import { GraphQLServer } from 'graphql-yoga'
// ... or using "require()"
// const { GraphQLServer } = require('graphql-yoga')
const typeDefs = `
  type Query {
    hello(name: String): String!
  }
`;
const resolvers = {
  Query: {
    hello: (_, { name }) => `Hello ${name || 'World'}`,
  },
};
const server = new GraphQLServer({ typeDefs, resolvers })
server.start(() => console.log("Server is running on localhost:4000"))
```

AYMEN SELLAOUTI

<https://github.com/prisma-labs/graphql-yoga>

23

GraphQL Implémentation



- En fournissant le typeDefs et le resolvers, `graphql-yoga` va construire l'instance `GraphQLSchema` en utilisant la fonction `makeExecutableSchema` de la bibliothèque `graphql-tools`.
- Une fois le serveur instancié vous pouvez le démarrer avec la méthode `start` qui prend en paramètre un objet `Options` et une `callback` lancé dès le démarrage du serveur.

```
import { GraphQLServer } from 'graphql-yoga'
// ... or using "require()"
// const { GraphQLServer } = require('graphql-yoga')
const typeDefs = `
  type Query {
    hello(name: String): String!
  }
`;
const resolvers = {
  Query: {
    hello: (_, { name }) => `Hello ${name || 'World'}`,
  },
};
const server = new GraphQLServer({ typeDefs, resolvers })
server.start(() => console.log("Server is running on localhost:4000"))
```

AYMEN SELLAOUTI

<https://github.com/prisma-labs/graphql-yoga>

24

GraphQL Implémentation



```
import { GraphQLServer } from 'graphql-yoga'
// ... or using "require()"
// const { GraphQLServer } = require('graphql-yoga')
const typeDefs = `
  type Query {
    hello(name: String): String!
  }
`;
const resolvers = {
  Query: {
    hello: (_, { name }) => `Hello ${name || 'World'} `,
  },
};
const server = new GraphQLServer({ typeDefs, resolvers })
server.start(() => console.log('Server is running on localhost:4000'))
```

Schema : Le contrat entre
GQL est l'utilisateur de l'API

Resolvers : L'implémentation
du contrat

AYMEN SELLAOUTI

<https://graphql.org/code/>

25

GraphQL Implémentation



- **typeDefs** est un argument obligatoire et doit être une chaîne contenant du langage de schéma GraphQL ou le path d'un fichier d'extension **'graphql'**

```
import { GraphQLServer } from 'graphql-yoga'

const server = new GraphQLServer(
  {
    typeDefs: "graphql/schema.graphql",
    resolvers
  })
server.start(() => console.log('Server is running on localhost:4000'))
```

AYMEN SELLAOUTI

<https://github.com/prisma-labs/graphql-yoga>

26

GraphQL Implémentation



- Afin de ne pas relancer le serveur à chaque modification, installer **nodemon** avec la commande `npm i nodemon`
- Au niveau de package.json définissez un script dans l'option script et spécifier l'utilisation de nodemon.
- Afin de spécifier à nodemon de prendre aussi en considération les modifications faites dans des extensions qu'il ne gère pas par défaut ajouter l'option `-ext ou -e` avec les extensions que vous voulez :

```
"scripts": {  
  "start": "nodemon --ext js,mjs,graphql index.mjs "  
},
```

AYMEN SELLAOUTI

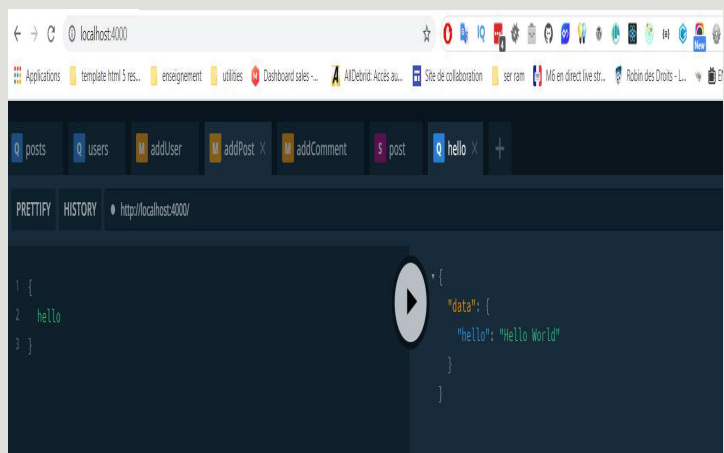
<https://www.npmjs.com/package/nodemon>

27

GraphQL Implémentation Playground



- Afin de requêter une API GraphQL,
- Vous pouvez utiliser l'outil playground qui vous offre une interface graphique.
- Lorsque vous lancer votre serveur, vous avez accès à cet outil sur le port 4000 qui est le port par défaut.




AYMEN SELLAOUTI

<https://graphql.org/code/>

28

GraphQL
Implémentation
Playground

 GraphQL

postsNew TabaddUseraddPostaddCommentpostfamilyMemebers

PRETTIFYHISTORYhttp://localhost:4000/

```
1 query familyMemebers($familyname: String!) {
2   infosUsers(name: $familyname) {
3     ...userInfos
4   }
5 }
6 fragment userInfos on User {
7   name
8   firstname
9 }
10
```

QUERY VARIABLESHTTP HEADERS

```
1 {
2   "familyname": "sellaouti"
3 }
```

Hitge

Search the docs ...

hello(name: String!): String!

TYPE DETAILS

hello(...): String!The String scalar type represents textual data, represented as UTF-8 character sequences. The String type is most often used by GraphQL to represent free-form human-readable text.

infos: User!


infosUsers(...): [User!]

scalar String

ARGUMENTS

name: String

AYMENSELLAOUTI29

 GraphQL

TYPES ET SCHEMA

AYMENSELLAOUTI30

GraphQL Schema



- Le Schema GraphQL définit les structures des requêtes valides offertes par votre serveur GraphQL ainsi que le type des valeur de retour.
- Ca représente le contrat entre le serveur et le client leur permettant de communiquer.
- C'est un **type fort** et qui permet d'utiliser les types scalaire ainsi que les objets et les énumérations.
- Il existe trois types d'opérations dans GraphQL qui sont les **query**, les **mutations** et les **subscription**.

AYMEN SELLAOUTI

31

GraphQL Schema



Schema

Root Query type

```
type Query {  
  allMedia: [Media]  
  firstMedia: Media  
  getMedia(id: Int!): Media  
  total: Int  
}
```

Return a list of Media

Argument required

Union of 2 types

```
union Media = Movie | Franchise
```

Object type

```
type Movie {  
  id: Int!  
  title: Title  
  genre: Genre  
}
```

Non-Nullable field

```
enum Category {  
  ADVENTURE  
  COMEDY  
  SCI_FI  
}
```

Enum type, used for options

Object type

```
type Title {  
  userPreferred: String!  
  original: String  
  variants: [TitleVariant]  
}
```

Built-in Scalar types:
String Int Float Boolean
and ID (not meant to be human-readable)

A Valid Query

Sub-selections MUST be provided for "Object" types

```
query {  
  allMedia {  
    id  
    title {  
      userPreferred  
    }  
  }  
  total  
}
```

NO sub-selections because they are "Scalar" types

@hexrsc

32

GraphQL Schema Definition Language(SDL) types



GraphQL

- GraphQL possède son propre système de typage qui permet de définir le schéma d'une API.
- La syntaxe permettant de le faire est appelée SDL

```
type user {  
  name : string  
  age: Int  
  isWorking: Boolean  
}
```

```
type Query { ... }  
type Mutation { ... }  
type Subscription { ... }
```

AYMEN SELLAOUTI <https://www.prisma.io/blog/graphql-sdl-schema-definition-language-6755bcb9ce51>

33

GraphQL Schema Definition Language types



GraphQL

- Un type est un **type de données** ou de **fonctionnalités** offertes par GraphQL.
- Un Type peut être un **scalaire**.
- Un type peut représenter un **objet du monde réelle** que vous manipuler.
- Il peut aussi représenter l'une des **opérations** prédéfinies et offertes par GraphQL et qui sont les entrées de l'utilisateurs et qui sont :
 - Query
 - Mutation
 - Subscription

```
type user {  
  name : string  
  age: Int  
  isWorking: Boolean  
}
```

```
type Query { ... }  
type Mutation { ... }  
type Subscription { ... }
```

AYMEN SELLAOUTI

34

GraphQL Schema Definition Language types – scalar types

Int A signed 32-bit integer

Float A signed double-precision floating-point value

String A UTF-8 character sequence

Boolean true or false values

ID Unique identifier. Used to re-fetch an object or as the key for a cache.

AYMENSELLAOUTI

35

GraphQL Schema Definition Language types - enums

Enumeration Types

```
enum language {  
  ENGLISH  
  SPANISH  
  FRENCH  
}
```

◀ Enums are special scalar types that are restricted to a particular set of allowed values.

AYMENSELLAOUTI

36

GraphQL Schema Definition Language types-Object



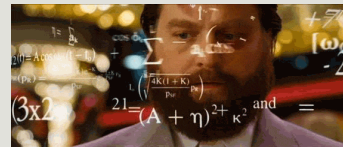
- Un objet est un type composite permettant de définir un objet du monde réel.
- Utiliser le mot clé `type` suivi du `nom de l'objet` suivie de `{}`.
- Définissez ensuite les propriétés dans l'objet avec un couple `clé valeur` identifiant pour la clé le `nom de la propriété` et pour la valeur son `type`.
- Le type peut être `scalaire` ou `composite`.

```
type user {  
  name : string  
  age: Int  
  isWorking: Boolean  
}
```

AYMEN SELLAOUTI

37

Exercice



- Définissez un type `todo`
- Un `todo` est caractérisé par un `id` de type `string`, un `name` et un `content` de type `string` et un `status` de type `enum`.
- Le `enum` `TodoStatus` contient les types suivants: `WAITING`, `IN_PROGRESS`, `CANCELED`, `DONE`
- Tous les champs sont obligatoires

AYMEN SELLAOUTI

38

GraphQL Schema Definition Language List et Non-Null



GraphQL

- Si vous voulez définir une liste d'un type donnée, il suffit d'utiliser les `[]` et mettre le type à l'intérieur.
- Pour spécifier qu'une valeur ne peut pas être nulle postfixer le type par `!`.
- En combinant les deux opérateurs, vous spécifier que la liste peut être nulle mais qu'elle ne peut pas contenir d'éléments null

```
type user {  
  name : string!  
  age: Int  
  roles: [Role]  
  isWorking: Boolean  
}
```

```
type user {  
  name : string!  
  age: Int  
  roles: [Role!]  
  isWorking: Boolean  
}
```

AYMENSELLAOUTI

39

GraphQL Schema Definition Language relation



GraphQL

- Afin de définir une relation entre deux objets, la syntaxe est très simple. Il suffit de créer une `propriété` dans l'un ou dans les deux objets du `type de l'autre objet`.

```
type Person {  
  name: String!  
  age: Int!  
  posts: [Post!]!  
}
```

```
type Post {  
  title: String!  
  author: Person!  
}
```

Person



Post

AYMENSELLAOUTI

40

GraphQL Core Resolvers Les opérations



- GraphQL offre trois opérations appelées resolvers :
 - **Query** : Permet de récupérer des données.
 - **Mutations** : Permet d'ajouter, modifier ou supprimer des données.
 - **Subscriptions** : permet d'être notifié en temps réel des modifications sur une ou plusieurs ressources.

AYMEN SELLAOUTI

41

GraphQL Core Resolvers Les opérations



- Un **resolver** est donc l'implémentation d'une route définie dans votre Schéma.
- Chaque méthode du **resolver** prend en paramètre 4 arguments :
 - **parent** : La route parent de ce resolver.
 - **args** : Les arguments envoyés à la route. C'est un objet JS.
 - **context** : objet partagé par tous les resolvers. On y met les informations à partager tels que l'utilisateur connecté ou l'accès à la base de données.
 - **info** : informations concernant la requête.

AYMEN SELLAOUTI

42

GraphQL Core

Resolvers

Les opérations

```
hello: (parent, { name }, context, info) => {  
  console.log('parent',parent);  
  console.log('context',context);  
  console.log('info',info);  
  return `Hello ${name || "World"}`;  
},
```

TERMINAL	PROBLEMS	OUTPUT	DEBUG CONSOLE
<pre>parent undefined context {} info { fieldName: 'hello', fieldNodes: [{ kind: 'Field', alias: undefined, name: [Object], arguments: [], directives: [], selectionSet: undefined, loc: [Object] }], returnType: String!, parentType: Query, path: { prev: undefined, key: 'hello' }, schema: GraphQLSchema {</pre>			



GraphQL

REQUÊTER GRAPHQL

GraphQL Core

Les opérations

Query : Définition



- Une **Query** permet de dire à GraphQL **quels sont les données à récupérer pour une route donnée**.
- Vous ne pouvez demander que les Querys définies dans le Schéma.
- Contrairement aux API REST, **vous définissez exactement ce que vous voulez** de l'ensemble des propriétés offertes.

AYMEN SELLAOUTI

45

GraphQL Core

Les opérations

Query : Définition



- Comme nous l'avons mentionné, pour fonctionner, un serveur GQL doit avoir le schéma définissant les routes ainsi que leur implémentation.
- Donc, pour définir une requête, vous devez toujours passer par deux étapes :
 - Définir dans votre schéma le contrat de la Query dans votre Schema en indiquant la route et ses paramètres.

```
type Query {  
  hello(name: String): String!  
}
```

- L'implémenter en définissons un type **Query** qui est un objet JS. Il prend en paramètre le nom de la route de type Query et comme valeur, la fonction à

```
Query: {  
  hello: (_, { name }) => `Hello ${name || 'World'}`,  
},
```

AYMEN SELLAOUTI

46

GraphQL Core

Les opérations

Query : Définition (Exemple)



```
const typeDefs = `
  type User {
    name: String
    firstname: String
  }
  type Query {
    hello(name: String): String!
    infos: User!
  }
`;
const resolvers = {
  Query: {
    hello: (_, { name }) => `Hello ${name || "World"}`,
    infos: () => ({ name: "sellaouti", firstname: "aymen" }),
  },
};
```

AYMEN SELLAOUTI

47

GraphQL Core

Les opérations

Query : Définition (Exemple)



- Afin d'améliorer la lisibilité et l'organisation de votre code, séparer vos différentes parties en des fichiers distincts ou chaque fichier se charge d'une seule tâche.
- Définissez votre schéma dans un fichier schema.graphql et créer un fichier par resolver. Donc créer un fichier Query.js contenant votre objet Query.

AYMEN SELLAOUTI

48

GraphQL Core

Les opérations

Query : Définition (Exemple)

```
export const Query = {  
  hello: (_, { name }) => `Hello ${name || "World"}`,  
};
```

Query.js

```
import { GraphQLServer } from "graphql-yoga";  
import { Query } from "../resolvers/Query.js";  
//Schema  
const typeDefs = "schema/schema.graphql";  
//Resolvers  
const resolvers = {  
  Query,  
};  
// starting server  
const server = new GraphQLServer({ typeDefs, resolvers });  
server.start(() => console.log("Server is running on localhost:4000"));
```

```
enum TodoEnum {  
  WAITING  
  IN_PROGRESS  
  CANCELED  
  DONE  
}  
type Todo {  
  id: ID!  
  name: String!  
  content: String!  
  status: TodoEnum!  
}  
type Query {  
  hello(name: String): String!  
}
```

schema.graphql

AYMEN SELLAOUTI

49

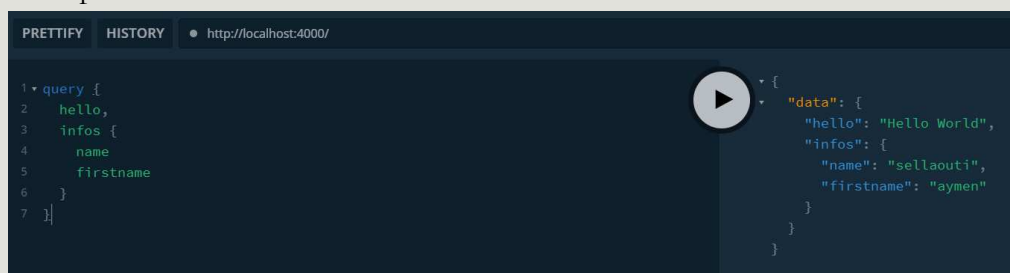
GraphQL Core

Les opérations

Query : Requête



- Une fois défini, vous pouvez accéder à votre Query via Playground en passant le mot clé `query` (optionnel) puis un objet `contenant la ou les routes demandées`.
- Pour `chaque route`, si la valeur de retour est de type `objet`, faite suivre le nom de la route d'un `objet` dans lequel vous définissez la liste des noms des champs à récupérer.



AYMEN SELLAOUTI

50

GraphQL Core Les opérations Query : Requête



- Si la valeur de retour est de type objet vous êtes obligé de définir les champs à récupérer. Sinon vous obtenez une erreur.

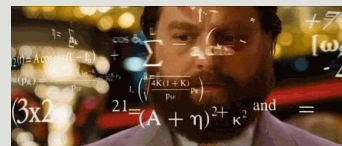
```
query {  
  hello,  
  infos  
}
```

```
{  
  "error": {  
    "errors": [  
      {  
        "message": "Field \"infos\" of type \"User!\" must have a  
        selection of subfields. Did you mean \"infos { ... }\"?",  
        "locations": [  
          {  
            "line": 3,  
            "column": 3  
          }  
        ]  
      }  
    ]  
  }  
}
```

AYMENSELLAOUTI

51

Exercice



- Créer un fichier 'bd.js'
- Ajouter y un tableau appelé todos contenant la liste des todos
- Exporter cet objet
- Créer une Query qui retourne la liste des Todos.

AYMENSELLAOUTI

52

GraphQL Core Les opérations Query : arguments



- Vous pouvez passer des arguments à vos Query.
- Dans votre schéma et lorsque vous implémenter votre route, définissez les arguments que vous attendez pour cette route.
- Un paramètre possède un **nom** et un **type** séparé par ‘:’
- Si vous voulez que ce paramètre soit **obligatoire** ajouter un ‘!’ devant le type.
- Lors de l’appel, passez les paramètres entre `()` en spécifiant le nom suivi de ‘:’ puis la valeur.
- Dans les deux cas séparer les paramètres par une **virgule**.

AYMENSELLAOUTI

53

GraphQL Core Les opérations Query : arguments



```
query {  
  getBooks(genre: COMIC, sort: ID, , ,) {  
    id  
    title  
    author {  
      name  
    }  
  }  
}  
  
const resolvers = {  
  Query: {  
    getMedia  
  }  
};  
  
function getMedia(root, args, context, info) {  
  const {genre, sort, count} = args;  
  // fetch, filter, sort ...  
  // put data into array of `Book`-shaped objects  
  return bookArray;  
}
```

Arguments

Commas, extra white spaces and line breaks are all ignored

Resolvers

Schema

Unused arguments default to values defined in the schema

```
type Query {  
  getBooks(genre: Genre!,  
    sort: Sort = POP,  
    count: Int = 10): [Book]  
}  
  
enum Sort {  
  POP  
  POP_ASC  
  ID  
  ID_ASC  
}  
  
enum Genre {  
  ADVENTURE  
  COMICS  
  FANTASY  
}  
  
type Book {  
  id: Int!  
  title: String  
  author: Author  
}  
  
type Author {  
  country: String  
  name: String  
}
```

@hexrcs

AYMENSELLAOUTI

54

GraphQL Core Query arguments : Exemple

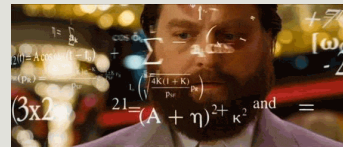


```
const users = [
  { name: "sellaouti", firstname: "aymen" },
  { name: "sellaouti", firstname: "skander" },
  { name: "Ben Slimane", firstname: "ahmed" },
];
type Query {
  hello(name: String): String!
  infos: User!
  infosUsers(name: String): [User!]
}
const resolvers = {
  Query: {
    infosUsers: (parent, args, ctx, info) => {
      return users.filter(
        (user) => user.name === args.name
      )
    },
  },
};
}
AYMEN SELLAOUTI
```

```
1 query {
2   infosUsers(name: "sellaouti") {firstname}
3 }
{
  "data": {
    "infosUsers": [
      {
        "firstname": "aymen"
      },
      {
        "firstname": "skander"
      }
    ]
  }
}
```

55

Exercice



- Ajouter une Query qui à partir de l'id d'un todo vous retourne le todo en question.

AYMEN SELLAOUTI

56

GraphQL Core

Les opérations

Query : alias



- Imaginer que vous voulez le résultat d'une même query mais avec des paramètres différents. Par exemple `infosUsers(name:'sellaouti')` et `infosUsers(name:'ben slimen')`.
- Ceci est impossible avec GQL sans l'utilisation des alias puisqu'on ne peut pas appeler la même ressource deux fois.
- La solution est l'utilisation des **Alias** avec la syntaxe suivante :
 - `aliasName : ressource`
 - `sellaoutiFamily : infosUsers(name:'sellaouti')`

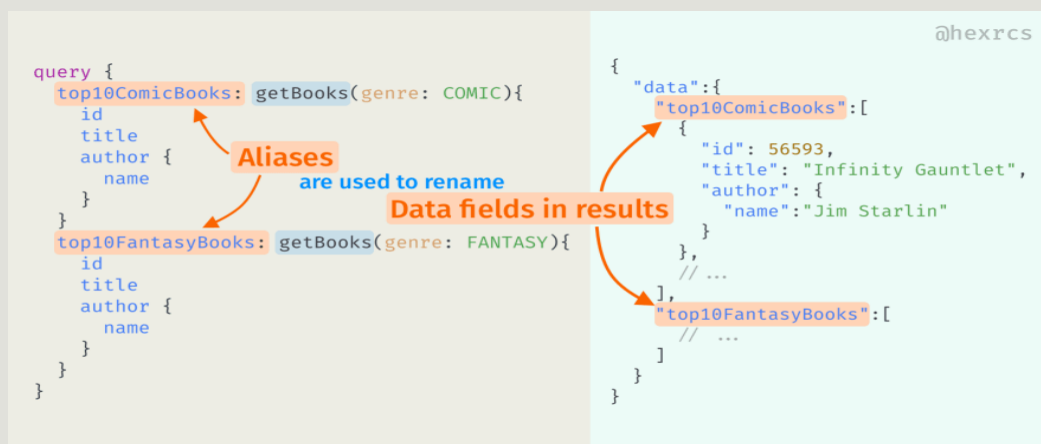
AYMENSELLAOUTI

57

GraphQL Core

Les opérations

Query : alias



AYMENSELLAOUTI

58

GraphQL Core Query alias : Exemple



```
1 query {  
2   infosUsers(name: "sellaouti") {firstname}  
3   infosUsers(name: "Ben Slimane") {firstname}  
4 }
```

```
{  
  "error": {  
    "errors": [  
      {  
        "message": "Fields \"infosUsers\"  
        conflict because they have differing  
        arguments. Use different aliases on the  
        fields to fetch both if this was  
        intentional.",  
      }  
    ]  
  }  
}
```

```
1 query {  
2   selloutiFamily: infosUsers(name: "sellaouti") {firstname}  
3   benSlimaneFamily: infosUsers(name: "Ben Slimane") {firstname}  
4 }
```

```
{  
  "data": {  
    "selloutiFamily": [  
      {  
        "firstname": "aymen"  
      },  
      {  
        "firstname": "skander"  
      }  
    ],  
    "benSlimaneFamily": [  
      {  
        "firstname": "ahmed"  
      }  
    ]  
  }  
}
```

AYMENSELLAOUTI

59

GraphQL Core Les opérations Query : fragments



- Dans plusieurs cas d'utilisations, vous allez vous retrouver à requêter les mêmes champs.
- Au lieu de réécrire à chaque fois la même chose vous pouvez utiliser des fragments.
- Un fragment est un ensemble de champs d'un objet.

```
fragment Name on TypeName {  
  champ1  
  champ2  
  ...  
  champ n  
}
```

AYMENSELLAOUTI

60

GraphQL Core

Les opérations

Query : fragments



```
query {
  top10ComicBooks: getBooks(genre: COMIC){
    ... BookGeneralInfo
  }
  top10FantasyBooks: getBooks(genre: FANTASY){
    ... BookGeneralInfo
    author {
      name
    }
  }
}

fragment BookGeneralInfo on Book {
  id
  title
}
```

Use the fragment with 3 dots

Additional selection fields

Define a fragment on a certain type

The fragment can be used here because the return type is Book

@hexrcs

GraphQL Core

Les opérations

Query : fragments



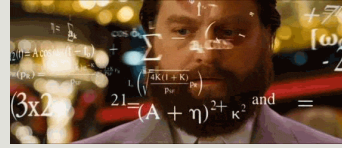
```
query {
  infosUsers(name: "sellaouti") {
    ...userInfos
  }
}

fragment userInfos on User {
  name
  firstname
}
```

▶

```
{
  "data": {
    "infosUsers": [
      {
        "name": "sellaouti",
        "firstname": "aymen"
      },
      {
        "name": "sellaouti",
        "firstname": "skander"
      }
    ]
  }
}
```

Exercice



- Définissez au niveau de playground un fragment avec l'ensemble des propriétés d'un todo afin de vous faciliter la récupération de toutes les informations d'un todo.

GraphQL Core

Les opérations

Query : variables



- Dans plusieurs cas d'utilisations, vous allez avoir des query paramétrables. Donc, au lieu d'avoir des constantes, vous devez définir vos requêtes.
- Commencer par nommer votre query et passez lui les variables.
- Afin de spécifier une variable préfixer votre variable par '\$'.
- Dans votre query, utiliser le même nom de variable précédé par \$ afin de la passer à votre méthode.
- Dans playground, en bas à gauche, vous pouvez spécifier les valeurs à donner à vos variables dans un objet.
- Vous pouvez donner une valeur par défaut à votre variable lors de sa définition en la lui affectant.

GraphQL Core

Les opérations

Query : variables



Define variables and default value

```
query ($genre: Genre = FANTASY, $count: Int) {  
  topBooks: getBooks(genre: $genre, count: $count){  
    id  
    title  
    author {  
      name  
    }  
  }  
}
```

The query will be serialized as a string and placed in the query field

```
{  
  "genre": "ANIME",  
  "count": 10  
}
```

HTTP Reqeust

Request URL:
https://api.example.org/

Request Method: POST

Request Payload:

```
{  
  "query": "...",  
  "variables": { ... }  
}
```

Variables will be a plain JSON object

AYMENSELLAOUTI

65

GraphQL Core

Les opérations

Query : variables



PRETTIFY HISTORY ● http://localhost:4000/

```
1 query familyMemebers($familyname: String!) {  
2   infosUsers(name: $familyname) {  
3     ...userInfo  
4   }  
5 }  
6 fragment userInfo on User {  
7   name  
8   firstname  
9 }  
10
```

QUERY VARIABLES HTTP HEADERS

```
1 {  
2   "familyname": "sellaouti"  
3 }
```

▶

```
{  
  "data": {  
    "infosUsers": [  
      {  
        "name": "sellaouti",  
        "firstname": "aymen"  
      },  
      {  
        "name": "sellaouti",  
        "firstname": "skander"  
      }  
    ]  
  }  
}
```

AYMENSELLAOUTI

66

GraphQL Schema Definition Language relation



- Afin de définir une relation entre deux objets, la syntaxe est très simple. Il suffit de créer une **propriété** dans l'un ou dans les deux objets du **type de l'autre objet**.

```
type Person {  
  name: String!  
  age: Int!  
  posts: [Post!]!  
}  
  
type Post {  
  title: String!  
  author: Person!  
}
```



AYMENSELLAOUTI

67

GraphQL Schema Definition Language relation



- Dans ce cas d'utilisation, lorsque vous allez chercher les posts d'une personne, vous devez faire un traitement particulier. Ce traitement consiste à aller chercher les posts dont l'id author est celui de votre personne.
- Dans ce cas vous devez définir la query posts dans l'objet Person

```
type Person {  
  name: String!  
  age: Int!  
  posts: [Post!]!  
}  
  
type Post {  
  title: String!  
  author: Person!  
}
```



AYMENSELLAOUTI

68

GraphQL Schema Definition Language relation



- Comme vous l'avez fait pour le Query, définissez un objet

```
export const Post = {  
  author: (parent, args, context, infos) => {  
    //parent contient l'objet Post que vous avez récupérer  
    // récupérer de parent ce que vous voulez et faites le  
    // traitement nécessaire pour retrouver le author  
  },  
};
```

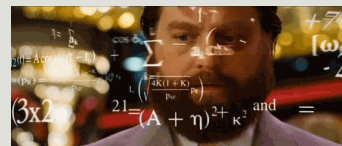
```
type Person {  
  name: String!  
  age: Int!  
  posts: [Post!]!  
}  
  
type Post {  
  title: String!  
  author: Person!  
}
```



AYMEN SELLAOUTI

69

Exercice



- Créer un type user avec un id, un name et un email.
- Chaque user peut avoir plusieurs todo et un todo est spécifique à un seul user.
- Faites le nécessaires pour pouvoir récupérer les todos d'un user et le user d'un todo.

AYMEN SELLAOUTI

70

GraphQL context

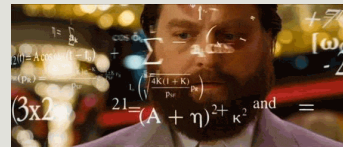


- Dans GraphQL, un contexte est un objet partagé par tous les resolvers d'une exécution spécifique.
- Il est utile pour conserver des données telles que les informations d'authentification, l'utilisateur actuel, la connexion à la base de données, les sources de données et d'autres éléments dont vous avez besoin pour exécuter votre logique métier.
- Le contexte est disponible comme 3e argument de chaque resolver.

AYMEN SELLAOUTI

71

Exercice



- Comme nous l'avons mentionné, chaque méthode du resolver prend en 3ème paramètre le context.
- Le **context** étant un objet partagé par tout les resolvers. Stocker y une fois pour tout votre objet db au lieu de le récupérer à chaque fois et passer le comme paramètre du **constructeur** de votre **GraphQLServer**.

AYMEN SELLAOUTI

72



TYPES AVANCÉES

AYMEN SELLAOUTI

73

GraphQL Schema Definition Language Les interfaces



- Une interface est un type abstrait contenant un ensemble de types.
- Chaque type implémentant cette interface doit obligatoirement contenir ces types.
- Pour définir une interface, utiliser le mot clé interface suivi du nom de l'interface puis l'objet que définit l'interface.

```
interface userInfos {  
  name : string!  
  email: string!  
}
```

AYMEN SELLAOUTI

74

GraphQL Schema Definition Language Les interfaces



- Les interfaces permettent aussi d'avoir du code plus flexible et polymorphe.
- Imaginer que vous voulez retourner un tableau d'utilisateurs qu'ils soient de type User ou Client.. Il suffit de définir une Query qui retourne des UserInfos.
- D'un autre côté vous voulez aussi récupérer les infos propres au User et au Client. Les interfaces nous permettent de faire ça.

```
type Human implements Character {  
  id: ID!  
  name: String!  
  appearsIn: [Episode]!  
  totalCredits: Int  
}
```

```
type Droid implements Character {  
  id: ID!  
  name: String!  
  appearsIn: [Episode]!  
  primaryFunction: String  
}
```

```
interface Character {  
  id: ID!  
  name: String!  
  appearsIn: [Episode]!  
}
```

AYMEN SELLAOUTI

75

GraphQL Schema Definition Language Les interfaces



```
interface UserInfos {  
  name: String!  
  email: String!  
}
```

```
type User implements UserInfos {  
  name: String!  
  email: String!  
  firstname: String  
}
```

```
type Client implements UserInfos {  
  name: String!  
  email: String!  
  bonusPoints: Int  
}
```

- Il faudra aussi définir la ressource UserInfos au niveau de votre resolver. En effet, graphql ne peut pas savoir quel objet récupérer parmi ceux qui implémentent l'interface.
- Pour ce faire au niveau de vos resolvers, implémenter la ressource UserInterface et développer la méthode __resolveType qui prend en paramètre l'objet à gérer.
- L'astuce est de tester sur l'existence de l'un des attributs qui permet de différencier l'objet.

AYMEN SELLAOUTI

76

GraphQL Schema Definition Language Les interfaces



```
const resolvers = {  
  > Query: { ...  
    },  
  > Mutation: { ...  
    },  
  UserInfos: {  
    __resolveType(obj) {  
      if (obj.firstname) {  
        return "User";  
      } else if (obj.bonusPoints) {  
        return "Client";  
      }  
    },  
  },  
};
```

AYMEN SELLAOUTI

77

GraphQL Schema Definition Language Les interfaces



- Ensuite pour récupérer les infos propre à un des deux types, on utilise les inline fragments.
- La syntaxe est la suivante :
... on **ObjectName** {
 propriété 1
 propriété 2
 ...
 propriété n
}

```
1  
2 query familyMemebers($familyname: String!) {  
3   infosUsers(name: $familyname) {  
4     ...userInfos  
5     ... on Client {bonusPoints}  
6     ... on User {firstname}  
7   }  
8 }  
9 fragment userInfos on UserInfos {  
10  name  
11  email  
12 }  
13
```

AYMEN SELLAOUTI

78

GraphQL

Schema Definition Language

Les interfaces



```
1 query familyMemebers($familyname: String!) {
2   infosUsers(name: $familyname) {
3     ...userInfos
4     ... on Client {bonusPoints}
5     ... on User {firstname}
6   }
7 }
8
9 fragment userInfos on UserInfos {
10   name
11   email
12 }
13
```

QUERY VARIABLES HTTP HEADERS

```
1 {
2   "familyname": "sellaouti"
3 }
```

```
{
  "data": {
    "infosUsers": [
      {
        "name": "sellaouti",
        "email": "aymen.sellaouti@techwall.tn",
        "firstname": "aymen"
      },
      {
        "name": "sellaouti",
        "email": "aymen.sellaouti@techwall.tn",
        "bonusPoints": 5
      }
    ]
  }
}
```

DOCS

SCHEMA

```
#
type Query {
#
  hello(name: String): String!
#
  infos: User!
#
  infosUsers(name: String): [UserInfos!]
#
}

type User implements UserInfos {
#
  name: String!
#
  firstname: String
#
  email: String!
#
}
```

GraphQL

Schema Definition Language

Les interfaces



Schema

```
interface Media {
  id: Int!
  title: String!
}

interface Series {
  id: Int!
  episodeCount: Int
  isRunning: Boolean
}

type Movie implements Media {
  id: Int!
  title: String!
}

type TVShow implements Media, Series {
  id: Int!
  title: String!
  episodeCount: Int
  isRunning: Boolean
}
```

We can implement multiple interfaces

Query

```
query {
  getMedia {
    id
    title
    ... on TVShow {
      isRunning
    }
  }
}
```

The sub-selection isRunning will only be performed when the Media is also TVShow

@hexrcs

GraphQL Schema Definition Language Les unions



- Les unions est une union de types. Il permet de créer un Type méta permettant de requêter sur différents types Objet en même temps.
- Imaginons qu'on réalise une fonctionnalité sur le site qui cherche tous les éléments dont le nom ou la désignation contient le mot recherché.

Syntaxe : **union** nomDeLUnion = **Type1** | **Type2** | ... | **TypeN**

- Comme pour les unions, Il faudra aussi définir la ressource de votre union. Pour ce faire au niveau de vos resolvers, implémenter la ressource et développer la méthode `__resolveType` qui prend en paramètre l'objet à gérer.

AYMEN SELLAOUTI

81

GraphQL Schema Definition Language Les unions



```
union searchObject = Client | User
```

```
type Query {  
  hello(name: String): String!  
  infos: User!  
  infosUsers(name: String): [UserInfos]!  
  search(name: String): [searchObject]!  
}
```

```
const resolvers = {  
  Query: {  
    hello: (parent, { name }, context, info) => { ... },  
    infos: () => ({ name: "sellaouti", firstname: "aymen" }),  
    infosUsers: (parent, args, ctx, info) => { ... },  
    search: (parent, { name }, context, info) => {  
      return users.filter(user => user.name === name);  
    },  
  },  
  Mutation: { ... },  
  UserInfos: { ... },  
  searchObject: {  
    __resolveType(obj) {  
      if (obj.firstname) {  
        return "User";  
      } else if (obj.bonusPoints) {  
        return "Client";  
      }  
    },  
  },  
};
```

AYMEN SELLAOUTI

82



LES MUTATIONS

AYMEN SELLAOUTI

83

GraphQL Core Les opérations mutations



- Les mutations permettent de gérer toute la partie persistance.
- Les opérations d'ajout, modification et suppression se font via les mutations.
- Les mutations ont la même syntaxe que les query mais en utilisant le mot clé mutation (lorsque vous faite votre requête) et le mot clé Mutation au niveau du schéma.
- Comme pour les query, nous pouvons utiliser les variables.

AYMEN SELLAOUTI

84

GraphQL Core Les opérations mutations



Schema

```
type Mutation {  
  createUser(input: UserInfoInput!): UserInfo  
  updateUser(id: Int!, input: UserInfoInput!): UserInfo  
}  
  
input UserInfoInput {  
  username: String!  
  password: String!  
}  
  
type UserInfo {  
  id: Int!  
  username: String  
  password: String  
}
```

Common practice is to wrap the arguments of an mutation in an input type

Query

Use mutation instead of query

```
mutation($input: UserInfoInput!) {  
  createAccount(input: $input) {  
    id  
  }  
}
```

There's always a return for a mutation

```
{  
  "input": {  
    "username": "AzureDiamond",  
    "password": "hunter2"  
  }  
}
```

Variables @hexracs

AYMEN SELLAOUTI

85

GraphQL Schema Definition Language Les input types.



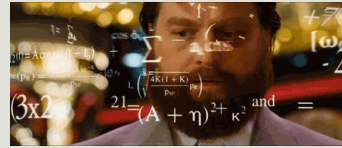
- Les input sont des types qui permettent de définir un objet décrivant les paramètres attendues comme argument évitant ainsi les types scalaires.
- Permettent de définir des types d'entrées de vos mutations (genre de DTO d'un seul côté)
- **Ne peuvent contenir que des types scalaires**
- Définit comme un Type sauf qu'on utilise le mot clé input

```
input AddUserInput {  
  name: String!  
  firstname: String!  
  email: String!  
}
```

AYMEN SELLAOUTI

86

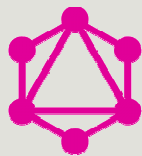
Exercice



- Créer deux inputs pour l'ajout et la modification d'un todo
- Pour ajouter un todo il faut un name, un content et un userId
- Pour modifier un todo, il faut optionnellement un name, un content, un userId et un status.
- Ajouter les mutations permettant d'ajouter ou de mettre à jour un todo.
- N'oublier pas de vérifier l'existence du user.
- Ajouter la mutation pour gérer la suppression.

AYMEN SELLAOUTI

87



GraphQL

SUBSCRIPTION

AYMEN SELLAOUTI

88

GraphQL Core

Les opérations subscriptions



- Les subscriptions sont une fonctionnalité GraphQL qui permet à un serveur d'envoyer des données à ses clients lorsqu'un événement spécifique se produit.
- Les subscriptions sont généralement implémentés avec WebSockets.
- Dans cette configuration, le serveur maintient une connexion stable avec son client abonné.

AYMEN SELLAOUTI

89

GraphQL Core

Les opérations subscriptions

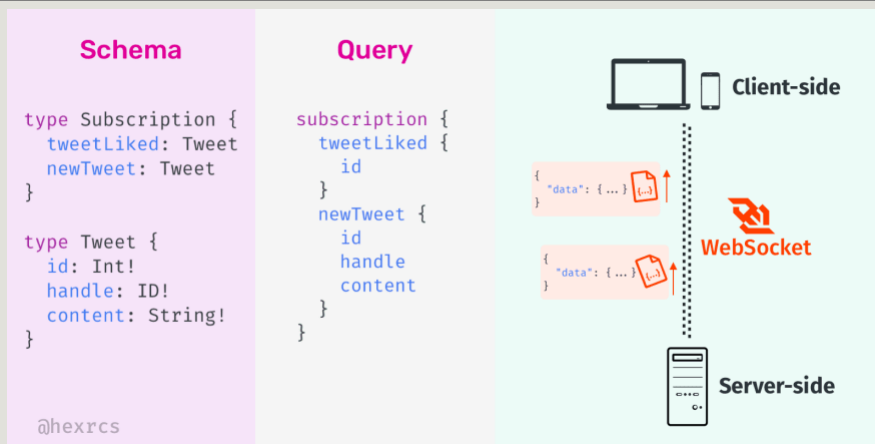


- Le client ouvre initialement une connexion au serveur en envoyant une requête d'abonnement qui spécifie l'événement qui l'intéresse.
- Chaque fois que cet événement particulier se produit, le serveur utilise la connexion pour transmettre les données d'événement au client abonné.

AYMEN SELLAOUTI

90

GraphQL Core Les opérations subscriptions



AYMEN SELLAOUTI

91

GraphQL Core Les opérations subscriptions



- Afin d'implémenter une subscription, vous pouvez utiliser la classe PubSub de GraphQLYoga. Cette classe est offerte par la bibliothèque graphql-subscriptions.
- importer cette classe et ajouter la dans votre context afin de pouvoir l'utiliser dans vos opérations.

```
import { PubSub } from "graphql-yoga";  
const pubsub = new PubSub();  
const server = new GraphQLServer({  
  typeDefs,  
  resolvers,  
  context: {  
    db,  
    pubsub  
  },  
});  
server.start() => console.log("Server is running on localhost:4000");
```

AYMEN SELLAOUTI

92

GraphQL Core

Les opérations subscriptions



- Maintenant que vous avez l'instance de la PubSub qui vous permettra de créer le tunnel entre le client et le serveur, identifier la subscription que vous souhaitez au niveau de votre schéma.

```
type subscription {  
  newTodo: Todo!  
}
```

AYMEN SELLAOUTI

93

GraphQL Core

Les opérations subscriptions



- Ensuite, allez-y et implémentez le *resolver* l'opération.
- Les *resolvers* pour les *subscriptions* sont légèrement différents de ceux des *Query* et des *Mutations*
 - Au lieu d'une méthode, vous allez passer un objet contenant une méthode *subscribe* qui récupère les paramètres habituels. .
 - Plutôt que de renvoyer des données directement, vous allez aller créer un **tunnel** et l'identifier via son **id**. Ceci se fait via un **AsyncIterator** qui est ensuite utilisé par le serveur GraphQL pour envoyer les données d'événement vers le client.

```
export const Subscription = {  
  newTodo: {  
    subscribe (parent, args, { pubSub }, info) {  
      return pubSub.asyncIterator('newTodo');  
    }  
  }  
}
```

AYMEN SELLAOUTI

<https://github.com/apollographql/graphql-subscriptions>

94

GraphQL Core

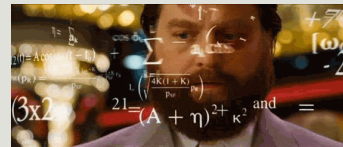
Les opérations subscriptions



- Une fois le tunnel crée, chaque entité qui **veut envoyer l'information** dans le tunnel devra récupérer l'objet **PubSub** et appeler la méthode **publish** en lui passant **l'identifiant du tunnel** ainsi que la partie **data** à envoyer.
- La partie data sera un objet avec comme clé le nom de la **subscription** et comme valeur la data à passer en paramètre.

```
addTodo: (parent, { addTodoInput }, { db, pubSub }, infos) => {  
  //...  
  pubSub.publish("newTodo", { newTodo : todo });  
  //...  
},
```

Exercice



- Créer une Subscription permettant de notifier sur l'ajout, la modification ou la suppression d'un Todo.