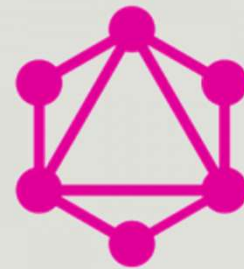


# GraphQL

---



GraphQL

AYMEN SELLAOUTI

[aymen.sellaouti@gmail.com](mailto:aymen.sellaouti@gmail.com)

# Introduction à GraphQL

---



GraphQL

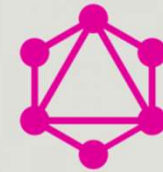
## Objectifs

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

# Introduction à GraphQL

## GraphQL un peu d'histoire

---



GraphQL

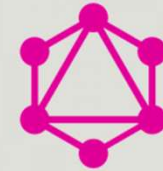
GraphQL a été développé en 2012 par FB. Les raisons qui ont poussé FB à chercher une alternative aux API REST sont les suivantes :

- 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'histoire

---

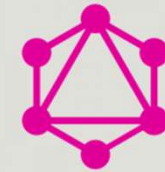


GraphQL

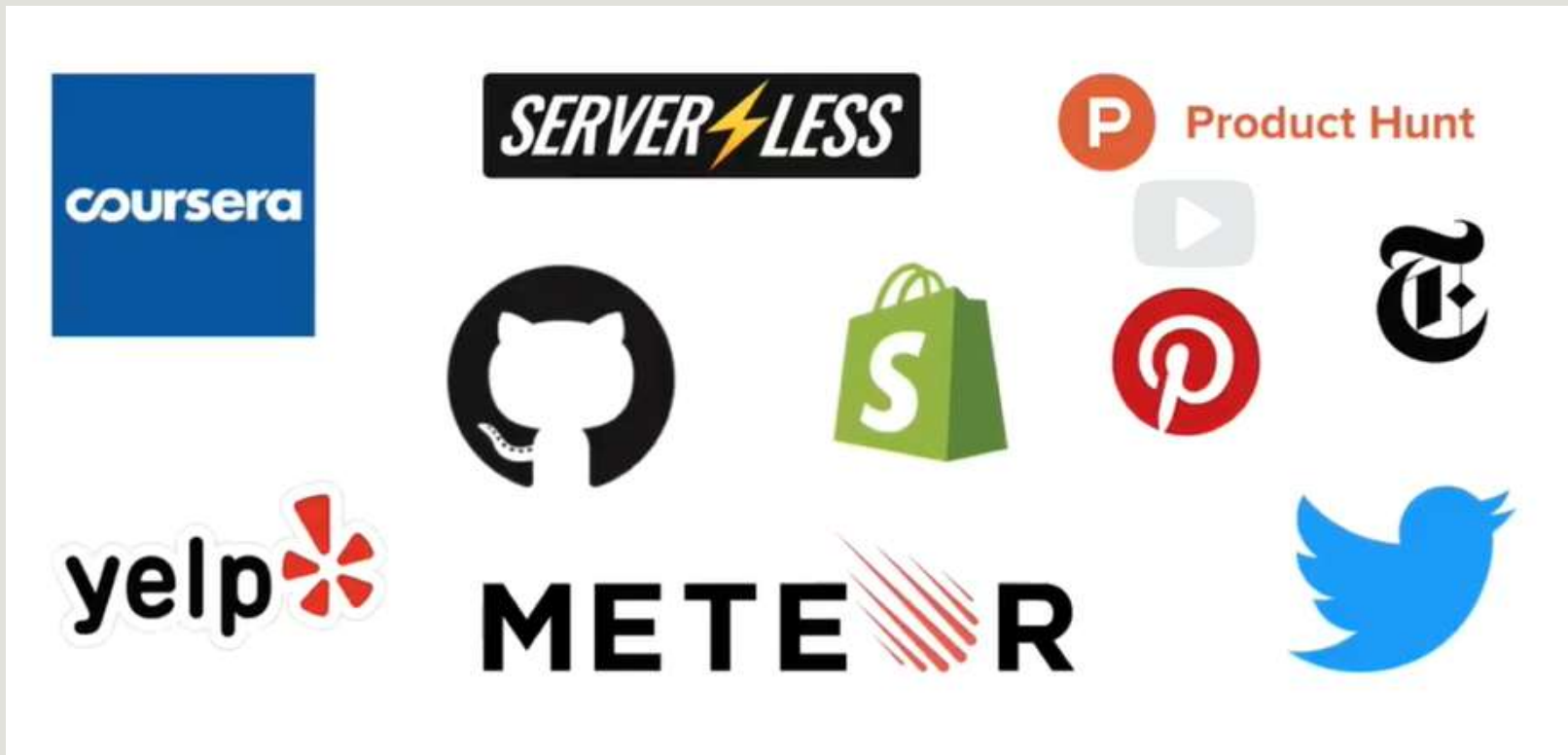
- GraphQL a été présenté au public en 2015 dans la conférence React.js.
- Il peut être utilisé par n'importe quel langage ou Framework
- D'autres entreprises tel que Coursera et Netflix (Falcor) ont entrepris des projets similaire qu'ils ont arrêtés à la sortie de GraphQL et l'ont utilisé.

# Introduction à GraphQL

## Ils ont choisi GraphQL



GraphQL



# Introduction à GraphQL

## Pourquoi GraphQL

---

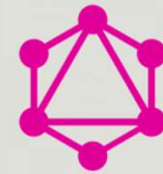


- Over fetching
  - Under fetching
- 
- Plus rapide : On peut regrouper plusieurs requêtes en une seule
  - Plus flexible : On peut demander ce qu'on veut pas besoin de spécifier coté serveur quoi retourner, c'est le client qui demande.

# Pourquoi GraphQL

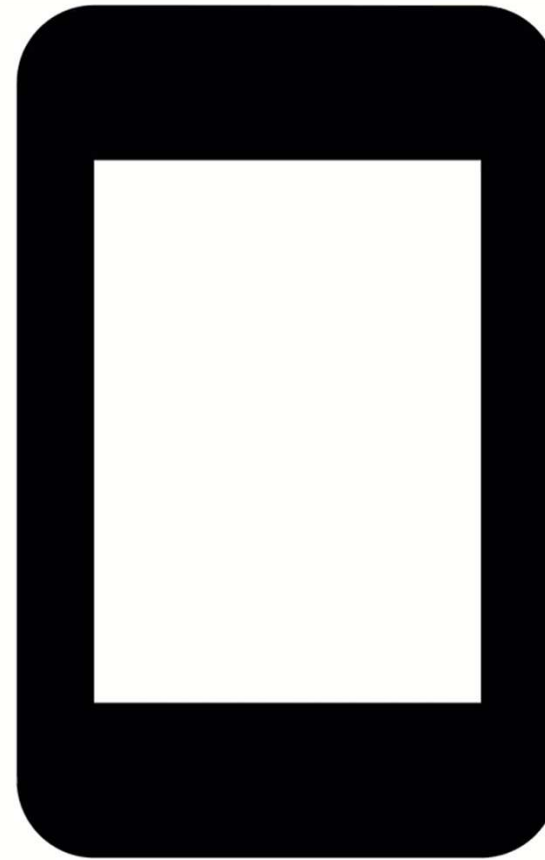
## GraphQL Vs REST

---



GraphQL

Example: **Blogging App**



# Pourquoi GraphQL

## GraphQL Vs REST

---



REST	GraphQL
Multiples 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

---



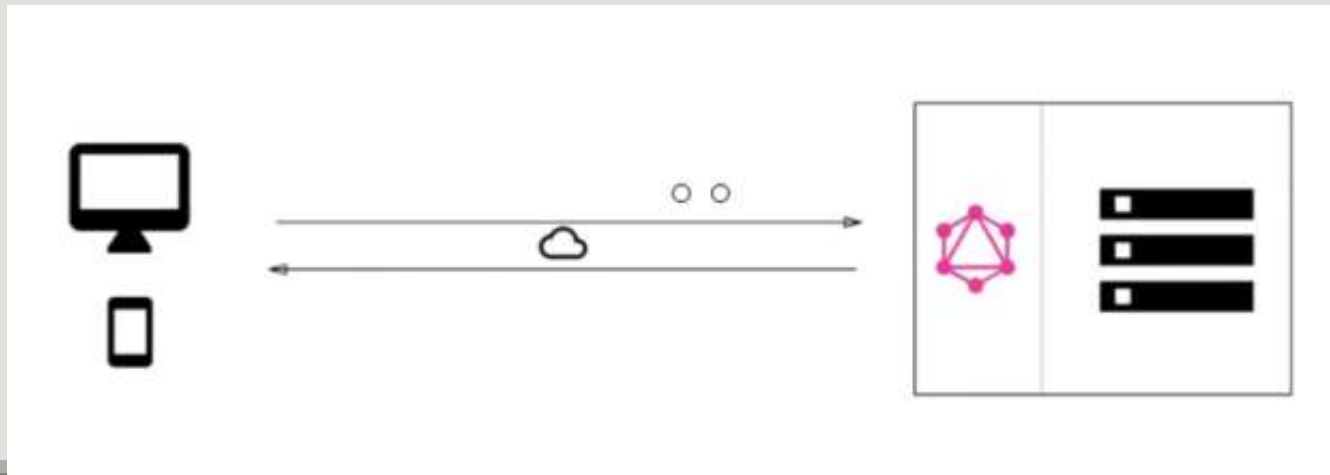
- GraphQL est une **spécification**. Il n'y a donc pas d'architecture particulière pour GraphQL.
- Cependant trois cas d'utilisations sont souvent rencontrés.
  - L'utilisation de GraphQL avec un **nouveau projet** d'une application **connecté à une BD**
  - L'intégration de GraphQL avec un **système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



# GraphQL Big Picture

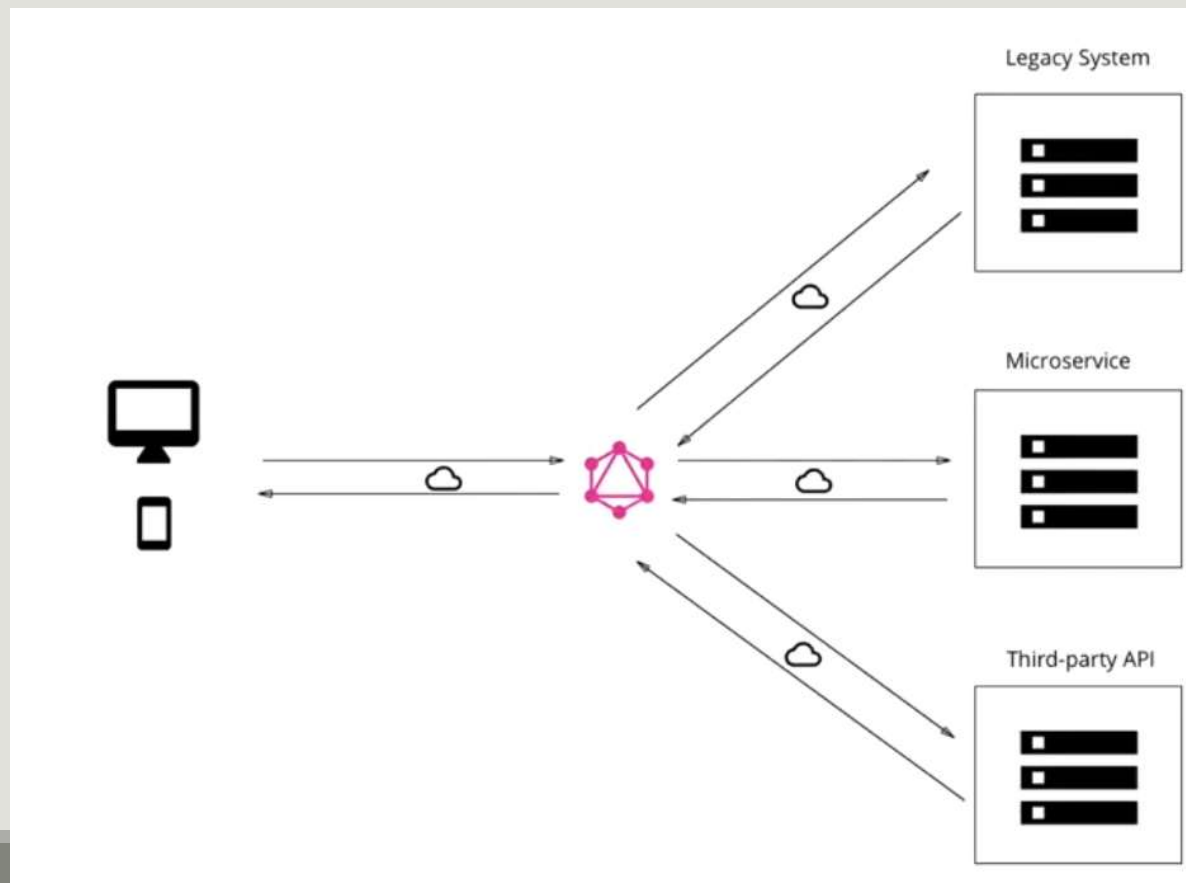
Intégrer GraphQL à un **système existant**



- 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.

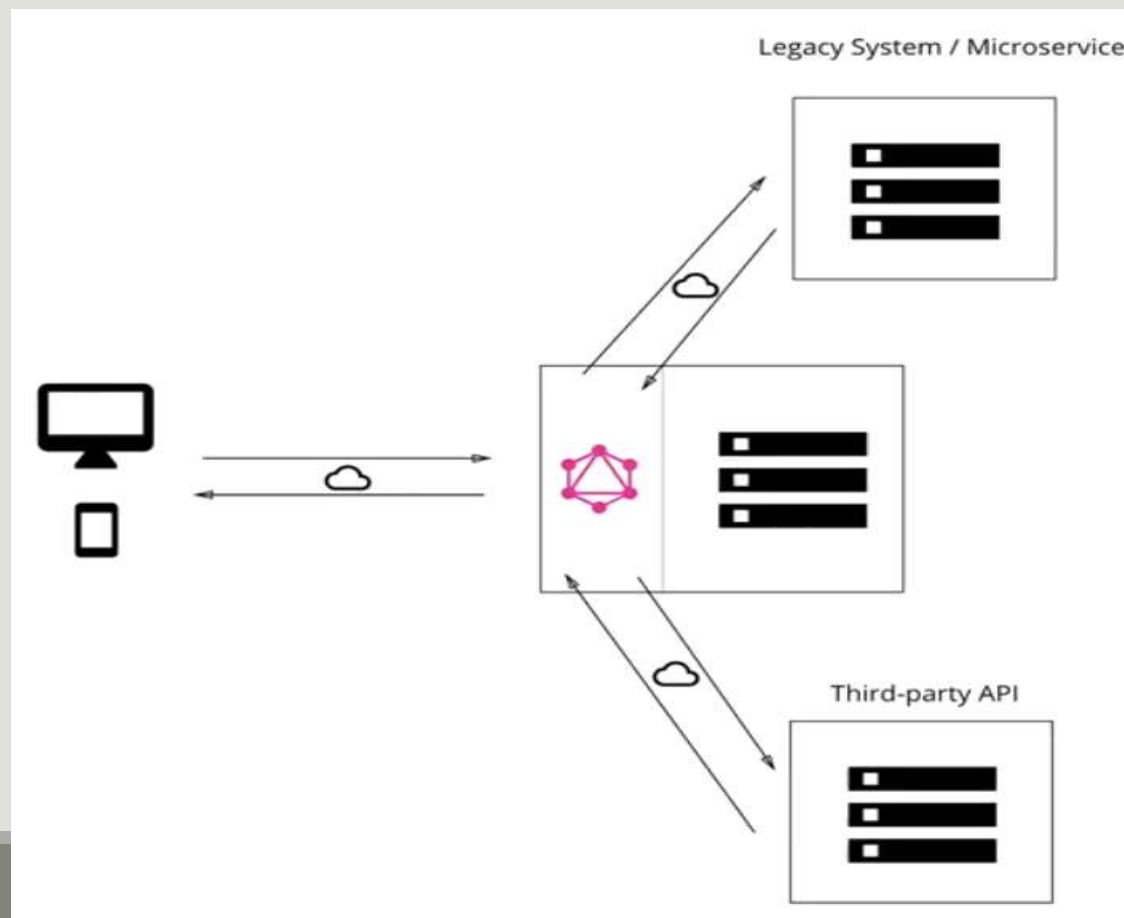
# GraphQL Big Picture

Intégrer GraphQL à un système existant

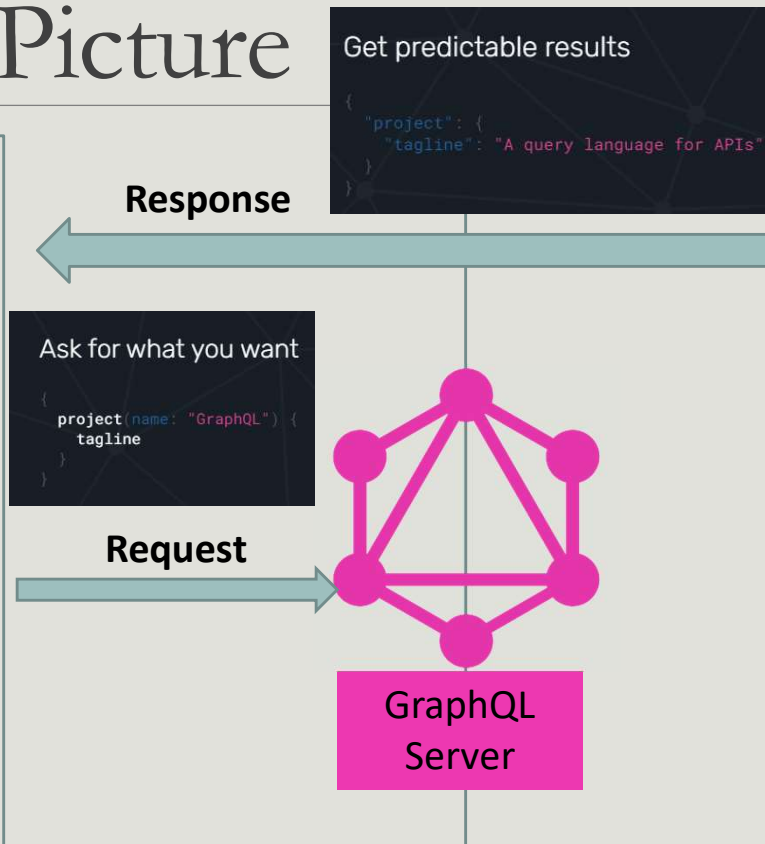
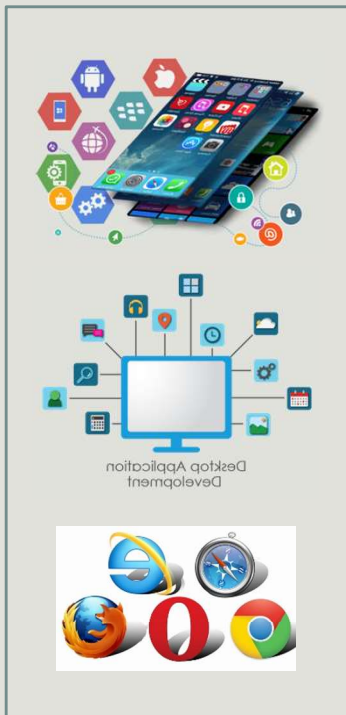


# GraphQL Big Picture

Système hybride



# GraphQL Big Picture



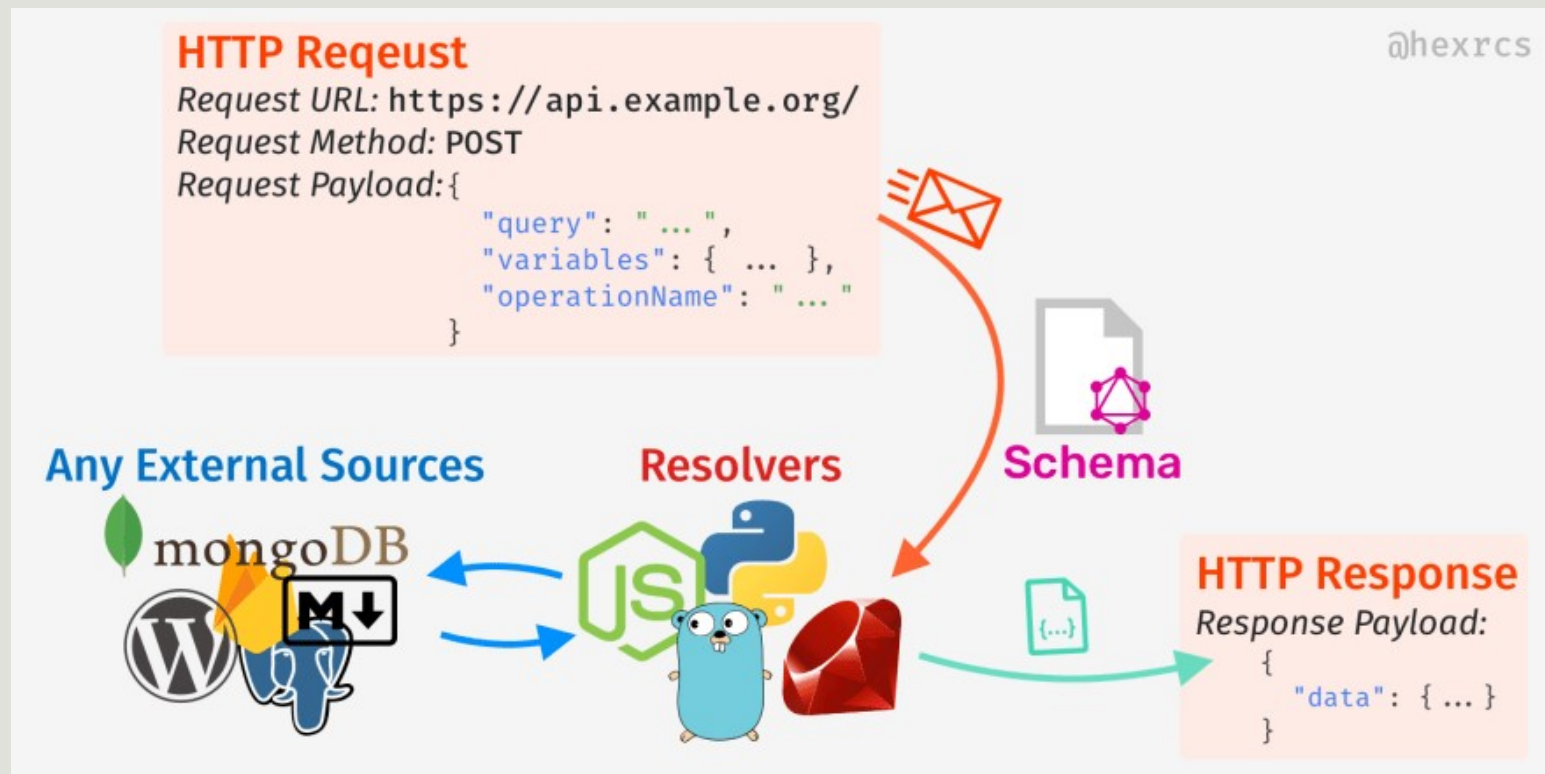
```
const resolvers = {
  Query: {
    hello: (_, { name }) => `Hello ${name || 'World'} `,
  },
};
export default resolvers;
```

```
schema {
  query: Query
  mutation: Mutation
}
```

Describe your data

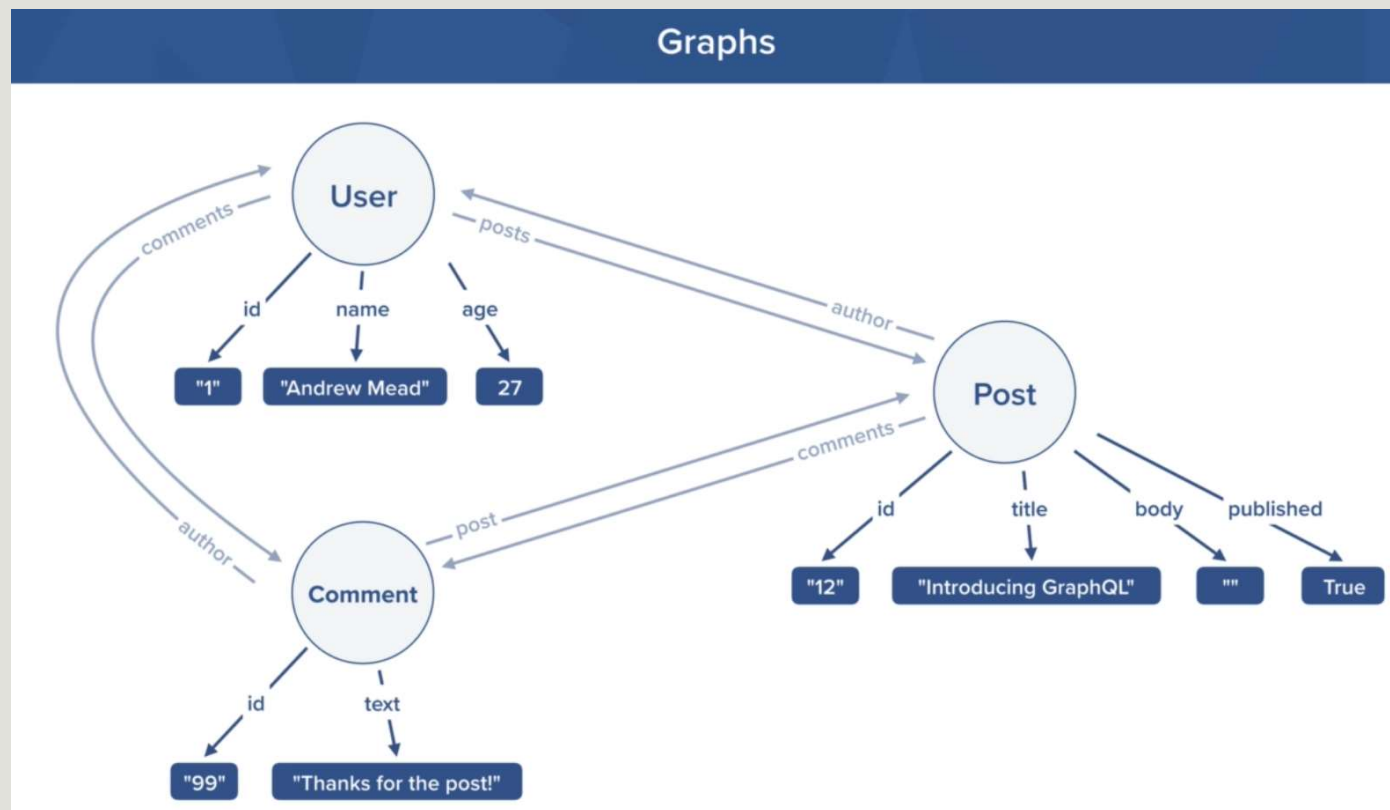
```
type Project {
  name: String!
  tagline: String!
  contributors: [User!]
}
```

# GraphQL Big Picture





# GraphQL Big Picture



# GraphQL Implémentation



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



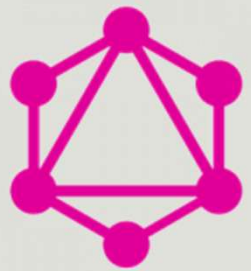
<https://github.com/prisma-labs/graphql-yoga>  
<https://the-guild.dev/graphql/yoga-server>

# GraphQL

## Cours Exemples

---

➤ <https://github.com/aymensellaouti/gqlG1325>



# GraphQL

---

IMPLÉMENTER VOTRE SERVEUR GRAPHQL

# GraphQL

## Implémentation



- On commence par préparer l'environnement  
npm i **typescript @types/node ts-node ts-node-dev cross-env**
- Initialisons package.json et la config typescript de notre projet :
  - Npm init --y
  - tsc --init
- Ajouter les scripts dev et start

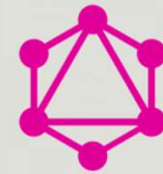
```
"scripts": {  
  "dev": "cross-env NODE_ENV=development ts-node-dev --exit-child --respawn src/main.ts",  
  "start": "ts-node src/main.ts"  
},
```

# GraphQL Implémentation



- Installer maintenant GraphQL Yoga :  
`npm install --save-exact graphql-yoga@3.9.0 @graphql-tools/schema graphql`
- Créer votre schema dans un fichier `schema.graphql`
- Créer votre resolver dans un fichier `Query.ts`
- Créer un fichier `main.ts` et ajouter y le code permettant de créer votre serveur GraphQL

# GraphQL Implémentation



GraphQL

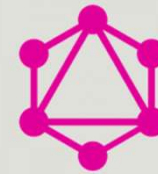
```
type Query {  
  hello: String!  
}
```

schema.graphql

```
export const Query = {  
  hello: () => 'Hello GL3'  
}
```

Query.ts

# GraphQL Implémentation



GraphQL

➤ **typeDefs** est un argument **obligatoire** et doit être de type **TypeSource** donc **une chaîne contenant du langage de schéma GraphQL**, le **path d'un fichier** d'extension **'graphql',...**

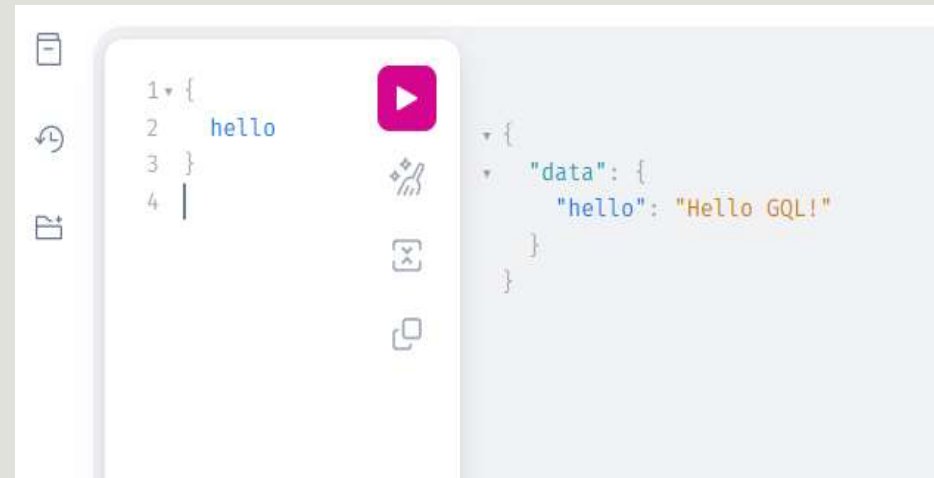
```
import { createSchema, createYoga } from "graphql-yoga";
import { createServer } from "http";
import { Query } from "../resolvers/Query";
const fs = require("fs");
const path = require("path");
export const schema = createSchema({
  typeDefs: fs.readFileSync(
    path.join(__dirname, "schema/schema.graphql"), "utf-8"
  ),
  resolvers: {
    Query
  },
});
function main() {
  const yoga = createYoga({ schema });
  const server = createServer(yoga);
  server.listen(4000, () => {
    console.info("Server is running on http://localhost:4000/graphql");
  });
}
main();
```



# GraphQL Implémentation Playground



- Afin de **requêter une API GQL**, 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.



# GraphQL Implémentation Playground



- Par défaut, le code GraphiQL est diffusé depuis un CDN, car si nous l'ajoutions à Yoga, le bundle serait volumineux et dépasserait la limite de charge utile pour certains environnements, par exemple (CF Workers, AWS, etc.). Si vous souhaitez utiliser GraphiQL depuis une version locale, vous devez l'installer manuellement.

```
npm i @graphql-yoga/render-graphiql
```

```
import { createYoga } from 'graphql-yoga'  
import { renderGraphiQL } from '@graphql-yoga/render-graphiql'  
const yoga = createYoga({ renderGraphiQL })
```

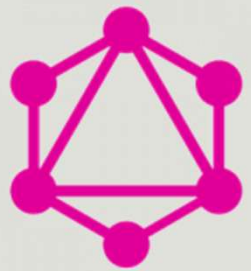
# GraphQL Implémentation Playground

A screenshot of the GraphQL Playground web application. The interface is divided into three main sections. On the left is a sidebar with a "Docs" tab selected, showing a document icon, a refresh button, and a list of "Root Types" including "query: Query". The middle section is a query editor with a line-numbered text area containing the query: 

```
1 {  
2   hello  
3 }  
4
```

. To the right of the query editor are icons for a play button, a schema icon, a variables icon, and a copy icon. The right section is a JSON viewer displaying the query result: 

```
{  
  "data": {  
    "hello": "Hello GQL!"  
  }  
}
```



# GraphQL

---

TYPES ET SCHEMA

# 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 typage 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.

# GraphQL Schema



# GraphQL

## Schema

Root Query type

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

Return a list of Media

Argument required

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

Union of 2 types

Object type

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

Non-Nullable field

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

Enum type,  
used for options

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

Object type

**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

@hexrcs

# 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 { ... }
```

# GraphQL

## Schema Definition Language

### types



- 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 manipulez.
- 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 { ... }
```



# 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.

# 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.

# GraphQL

## Schema Definition Language

### Les Objects



- Un objet est un type composite permettant de définir un objet du monde réel.
- Utilisez 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  
}
```

# 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 post fixer 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  
}
```

# 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!  
}
```



# 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 réels des modifications sur une ou plusieurs ressources.

# 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 parente de ce resolver.
  - **args** : Les arguments envoyée à la route. C'est un objet JS.
  - **context** : objet partagé par tout les resolvers. On y met les informations à partager tels que l'utilisateur connecté ou l'accès à la base de donnée.
  - **info** : informations concernant la requête.

# GraphQL Core

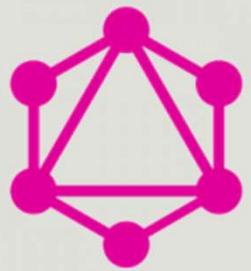
## Resolvers

### 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  
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 {
```





# 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.

# 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 typeDefs** en indiquant la route et ses paramètres.

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

# GraphQL Core

## Les opérations

### Query : Définition

---



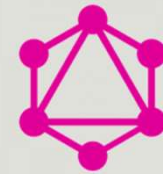
- L'implémenter en définissons un type `Query` qui est un objet JS. Il prend en paramètre le nom de la route de type `Query` et comme valeur, la fonction à exécuter.

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

# GraphQL Core

## Les opérations

### Query : Définition (Exemple)



GraphQL

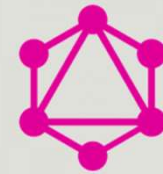
```
const typeDefs = `
  type User {
    name: String
    firstname: String
  }
  type Query {
    hello(name: String): String!
    infos: User!
  }
`;

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

# GraphQL Core

## Les opérations

### Query : Définition (Exemple)



GraphQL

- 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.

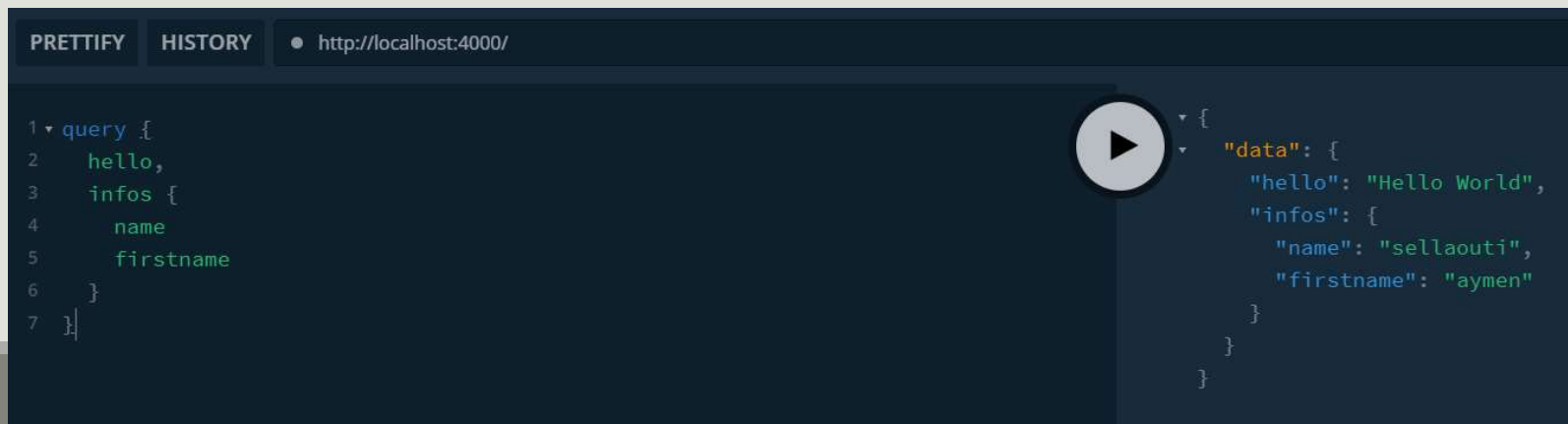
# GraphQL Core

## 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.

A screenshot of the GraphQL Playground interface. The top bar shows "PRETTIFY" and "HISTORY" buttons, and a URL "http://localhost:4000/". The left pane contains a query: 

```
1 query {  
2   hello,  
3   infos {  
4     name  
5     firstname  
6   }  
7 }
```

. The right pane shows the JSON response: 

```
{  
  "data": {  
    "hello": "Hello World",  
    "infos": {  
      "name": "sellaouti",  
      "firstname": "aymen"  
    }  
  }  
}
```

. A play button is visible between the two panes.

# 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  
          }  
        ]  
      }  
    ]  
  }  
}
```



# GraphQL Core

## Les opérations

### Query : Requête, Déclencher une erreur



- GraphQL vous fournit la classe `GraphQLError`.
- Elle prend en paramètre le message d'erreur et un objet d'options.

```
throw new GraphQLError('Element with id '${id}' not found.');
```

- Vous pouvez à travers l'objet **extensions** des options et son attribut **http** modifier le **status** de la réponse et ses **headers**.

```
throw new GraphQLError('Todo with id '${id}' not found.', {  
  extensions: {  
    http: {  
      status: 400,  
      headers: {  
        "x-custom-header": "some-value",  
      },  
    },  
  },  
});
```

# GraphQL Core

## Les opérations

### Query : arguments

---

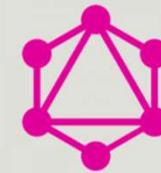


- Vous pouvez passer des arguments à vos Query.
- Dans votre schéma et lorsque vous implémenter votre route, définissez les arguments que vous attendez pour cette route.
- Un paramètre possède un **nom** et un **type** séparé par **':'**
- Si vous voulez que ce paramètre soit **obligatoire** ajouter 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**.

# GraphQL Core

## Les opérations

### Query : arguments



# GraphQL

```
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;  
}
```

## 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

# GraphQL Core

## Query arguments : Exemple



GraphQL

```
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"  
      }  
    ]  
  }  
}
```

# 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')**

# GraphQL Core

## Les opérations

### Query : alias



```
query {  
  top10ComicBooks: getBooks(genre: COMIC){  
    id  
    title  
    author {  
      name  
    }  
  }  
  top10FantasyBooks: getBooks(genre: FANTASY){  
    id  
    title  
    author {  
      name  
    }  
  }  
}
```

**Aliases**

are used to rename

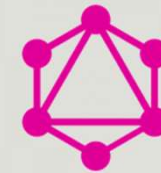
**Data fields in results**

@hexrcs

```
{  
  "data":{  
    "top10ComicBooks":[  
      {  
        "id": 56593,  
        "title": "Infinity Gauntlet",  
        "author": {  
          "name": "Jim Starlin"  
        }  
      },  
      // ...  
    ],  
    "top10FantasyBooks":[  
      // ...  
    ]  
  }  
}
```

# GraphQL Core

## Query alias : Exemple



GraphQL

```
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   sellayoutiFamily: infosUsers(name: "sellaouti") {firstname}  
3   benSlimaneFamily: infosUsers(name: "Ben Slimane") {firstname}  
4 }
```

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



# 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  
}
```



# GraphQL Core

## Les opérations

### Query : fragments



GraphQL

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

Use the fragment with 3 dots

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

Additional selection fields

```
fragment BookGeneralInfo on Book {  
  id  
  title  
}
```

Define a fragment on a certain type

@hexrcs

# GraphQL Core

## Les opérations

### Query : fragments



```
query {  
  infosUsers(name: "sellaouti") {  
    ...userInfos  
  }  
}  
  
fragment userInfos on User {  
  name  
  firstname  
}
```

A circular play button icon with a black triangle pointing to the right, centered on a dark blue background.

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

# 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 '\$'.

```
query($id: String = "1") {  
  
}
```

# GraphQL Core

## Les opérations

### Query : variables



- Dans vos requêtes, utilisez 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**.



```
query($identifiant: String = "1") {  
  user(id: $identifiant) {  
    name  
  }  
}
```

# 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 Request

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

*Request Method:* POST

*Request Payload:*

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

Variables will be a plain JSON object

@hexrcs

# GraphQL Core

## Les opérations

### Query : variables



```
PRETTIFY HISTORY ● http://localhost:4000/

1 query familyMembers($familyname: String!) {
2   infosUsers(name: $familyname) {
3     ...userInfos
4   }
5 }
6 fragment userInfos 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"
      }
    ]
  }
}
```

# 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!  
}
```



# 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!  
}
```

Person



Post



# 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 récupéré  
    // 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!  
}
```

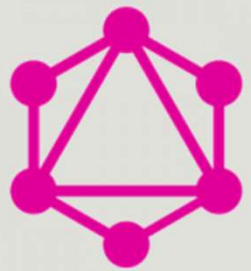


# GraphQL context



- 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 paramètre de l'objet passé à la méthode `createYoga`.

```
const yoga = createYoga({ schema, context: {db} });
```



# GraphQL

---

TYPES AVANCÉES

# 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 userInfo {  
  name : string!  
  email: string!  
}
```

# GraphQL

## Schema Definition Language

### Les interfaces



- Les interfaces permettent aussi d'avoir du **code plus flexible et polymorphique**.
- 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]!  
}
```

# GraphQL

## Schema Definition Language

### Les interfaces



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

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

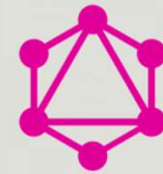
```
type Client implements UserInfos {  
  name: String!  
  email: String!  
  bonusPoints: Int  
}
```

- 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**.

# GraphQL

## Schema Definition Language

### Les interfaces



GraphQL

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

# GraphQL

## Schema Definition Language

### Les interfaces



GraphQL

- 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  
    }

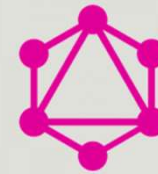
```
1
2 query familyMembers($familyname: String!) {
3   infosUsers(name: $familyname) {
4     ...userInfos
5     ... on Client {bonusPoints}
6     ... on User {firstname}
7   }
8 }
9 fragment userInfos on UserInfos {
10   name
11   email
12 }
13
```



# GraphQL

## Schema Definition Language

### Les interfaces



GraphQL

```
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
```

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



GraphQL

#### 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** sont des **unions de types**. Elles permettent 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.

# GraphQL

## Schema Definition Language

### Les unions

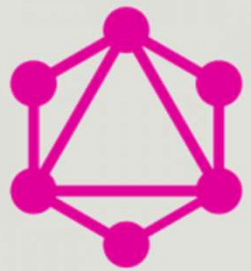


GraphQL

```
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";  
      }  
    },  
  },  
};
```



# GraphQL

---

LES MUTATIONS

# 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**

# GraphQL

## Schema Definition Language

### Les input types.



- Les **input** sont des **types** qui permettent de définir un objet décrivant les paramètres attendus 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!  
}
```



# GraphQL Core

## Les opérations mutations



# GraphQL

### 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  
}
```

Root Mutation

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  
  }  
}
```

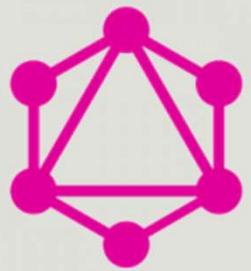
There's always a return for a mutation

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

Variables

@hexrcs





# GraphQL

---

SUBSCRIPTION

# 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.

# 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**.

# GraphQL Core

## Les opérations

### subscriptions



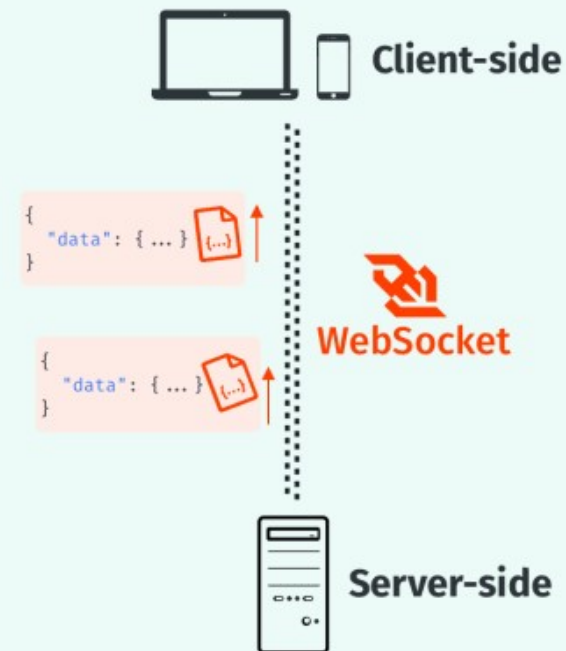
#### Schema

```
type Subscription {  
  tweetLiked: Tweet  
  newTweet: Tweet  
}  
  
type Tweet {  
  id: Int!  
  handle: ID!  
  content: String!  
}
```

@hexrcs

#### Query

```
subscription {  
  tweetLiked {  
    id  
  }  
  newTweet {  
    id  
    handle  
    content  
  }  
}
```



# GraphQL Core

## Les opérations subscriptions



- Afin d'implémenter une **subscription**, vous pouvez utiliser **createPubSub** de GraphQLYoga.
- Afin de la partager, vous pouvez l'ajouter dans le context.

```
function main() {  
  const pubSub = createPubSub();  
  const yoga = createYoga({  
    schema,  
    context: { db, pubSub },  
  });  
  const server = createServer(yoga);  
  server.listen(4000, () => {  
    console.info("Server is running on http://localhost:4000/graphql");  
  });  
}
```

# 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!  
}
```

# GraphQL Core

## Les opérations subscriptions



- 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 permet de s'inscrire à un Tunnel via son Id.
  - **resolve** qui prend en paramètre le payload de l'événement et retourne le résultat voulu aux subscribers.

```
export const Subscription = {  
  cvAdded: {  
    subscribe: (parent, args, { db, pubSub }) => pubSub.subscribe("cvAdded"),  
    resolve: (payload) => { return payload; },  
  },  
};
```

# 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.

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



---

aymen.sellaouti@gmail.com