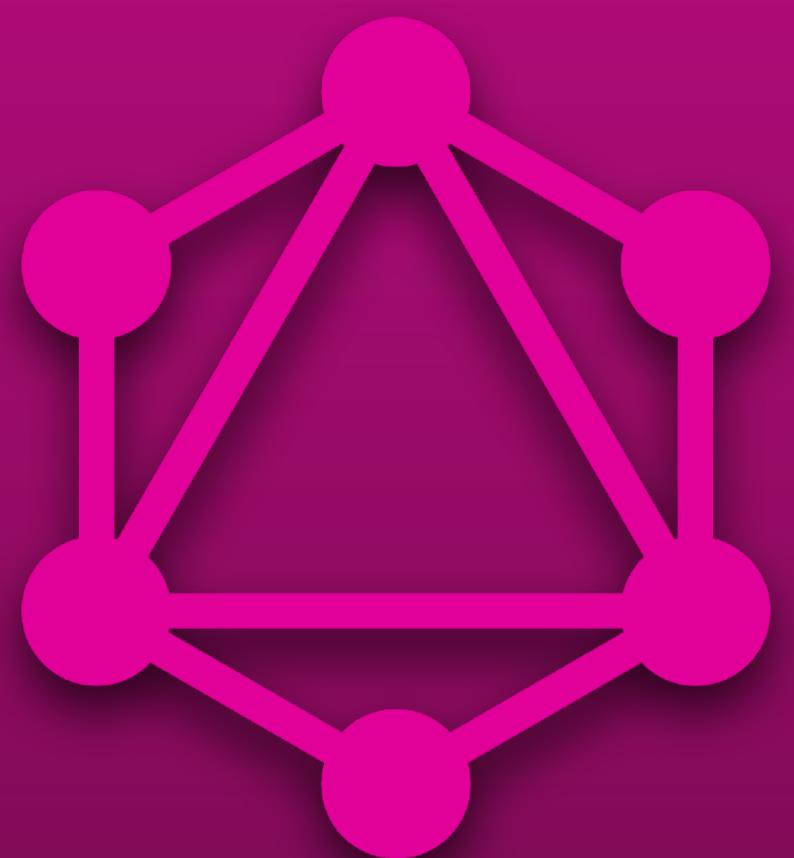
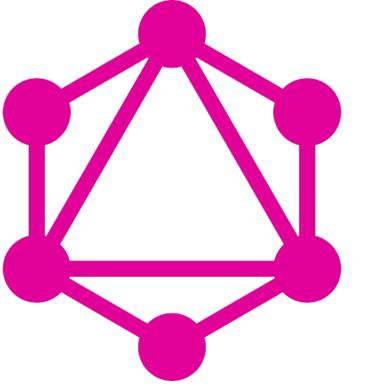


GraphQL

Key Concepts

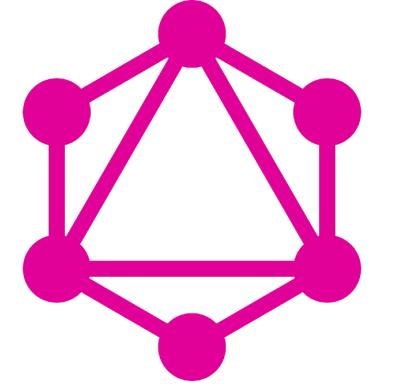


Au programme de ce cours

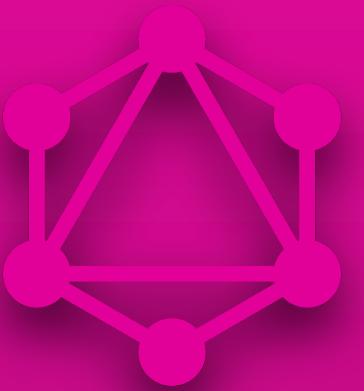


- Séance 1 : Fondamentaux et première API GraphQL
- Séance 2 : Schéma avancé et relations
- Séance 3 : Mutation, sécurité, client
- Séance 4 : Performance et temps réel
- Séance 5 : Projet final & Evaluation

Avant de commencer...

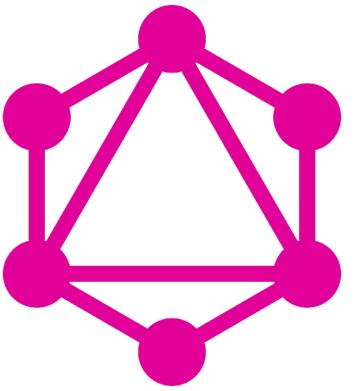


- Utilisation de l'intelligence artificielle générative **autorisé** mais j'aimerais qu'on passe un contrat tous ensemble :
 - Pour demander des explications supplémentaires ou un point de départ
 - Si on intègre des solutions données par l'IA, on teste, on challenge et on commente
 - Copier-coller les réponses sans comprendre réellement par flemme ou pour aller “plus vite”



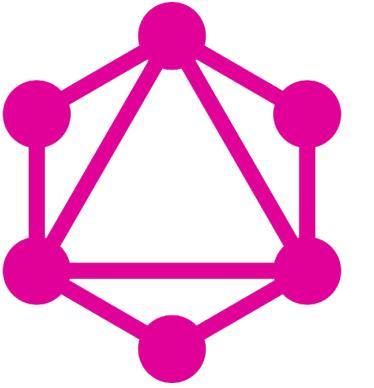
Séance 1 : Fondamentaux et première API GraphQL

De quoi allons-nous parler ?



- REST vs GraphQL
- Quelques définitions
- SDL et schéma minimal
- Un peu de pratique
- Un projet

REST vs GraphQL

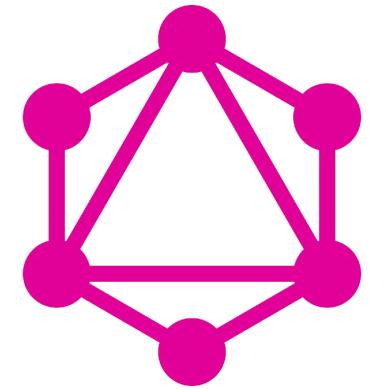


Les limites de REST

- Overfetching : On récupère trop de données, mais on en consomme qu'une partie
- Underfetching : on doit faire plusieurs requêtes pour construire une vue
- Une API par ressources -> fragmentation des appels et du code

REST vs GraphQL

Comparaison REST/GraphQL



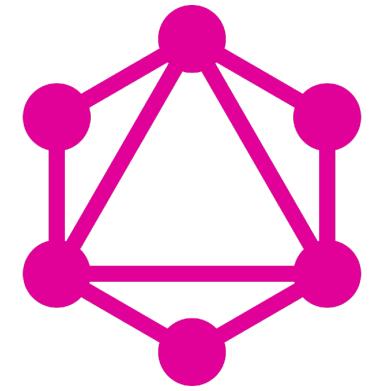
REST

- Plusieurs endpoints
- Structuration autour des ressources
- Risque de sur/sous-chargements

GraphQL

- Un seul endpoint
- Schéma typé (Query/
Mutation/Subscriptions)
- Récupère exactement les
champs nécessaires

Quelques définitions



Schéma

Contrat : définit les types,
champs et relations

Query

Lire les données (

- <=> Read (**CRUD**)
- <=> GET

)

Mutation

Modifier les données (

- <=> Create/Update/Delete (**CRUD**)
- <=> POST / PATCH / DELETE

)

Subscription

Recevoir les données en
temps réel

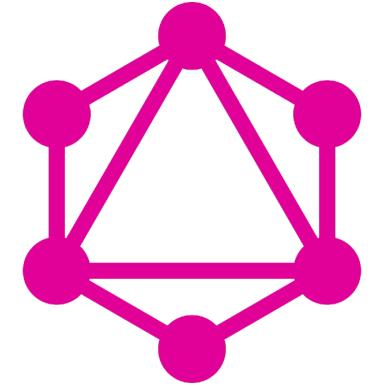
Resolver

Interface entre les Query/
Mutation et les logiques
métiers



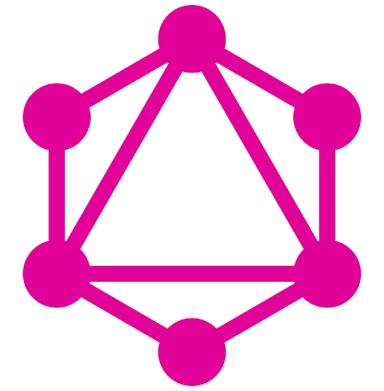
Question : Que se passe-t-il si je demande un champ qui n'existe pas dans le schéma ?

SDL et schéma minimal



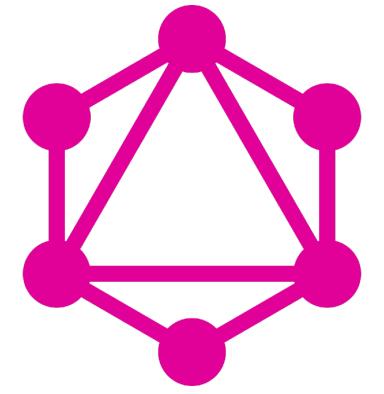
- Le schéma en SDL (Schema Definition Language) décrit les types disponibles et la manière de les relier. C'est un contrat fort.
- Le type **Query** est la porte d'entrée.
- Les types non exposés par Query ne sont pas accessibles depuis l'extérieur.
- Types et nullabilité : GraphQL possède des types scalaires (String, Int, Float, Boolean, ID) et des types objets. Les listes sont notées [Type]. On peut rendre un champ ou un élément non nul avec !. Par exemple : [String!]! signifie « liste non nulle de chaînes non nulles »

Un peu de pratique



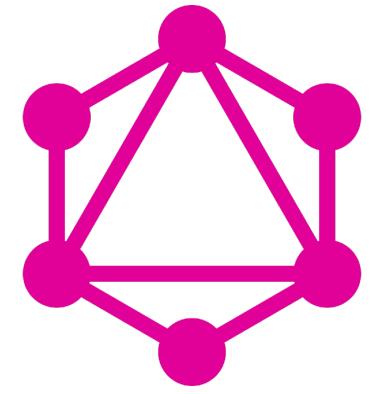
- Pour vous permettre de commencer à pratiquer nous allons partir à la découverte de **Playground**, la sandbox d'Apollo.
- L'objectif est de jouer avec l'**API GraphQL de Github** pour récupérer un ensemble de données (vous y trouverez la documentation ici : <https://docs.github.com/fr/graphql>).
- Une petite aide : <https://docs.github.com/fr/graphql/guides/using-graphql-clients>.

Un projet



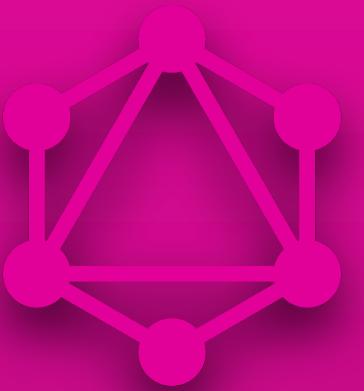
- Pour l'ensemble des séances, je vous propose un mini-projet vous permettant de mettre en pratique au fur et à mesure les différents concepts. L'objectif de ce mini projet est de construire un **gestionnaire d'évènements étudiants**. Pour cela vous avez le choix entre le faire en **JS** ou en **Python**.
- Si vous utilisez **JS** : Le projet se fera avec NodeJS et **Apollo GraphQL**
- Si vous utilisez **Python** : Le projet se fera avec **Graphene**
- **Objectif** : Créer un schéma minimal avec les types **User(id, name)** et **Event(id, title, date, organizer: User)**. Implémenter les resolvers en JS ou en Python pour renvoyer des données fictives. Interroger l'API pour récupérer des événements et leur organisateur.

Un projet



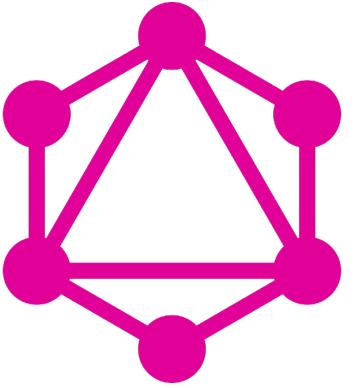
- Vous trouverez une proposition de correction de cette première séance ici :

<https://github.com/othila-academy/graphql-course>



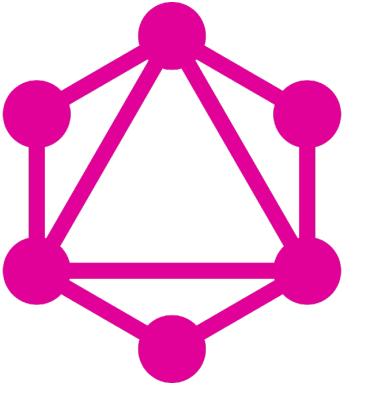
Séance 2 : Schéma avancé et relation

De quoi allons-nous parler ?



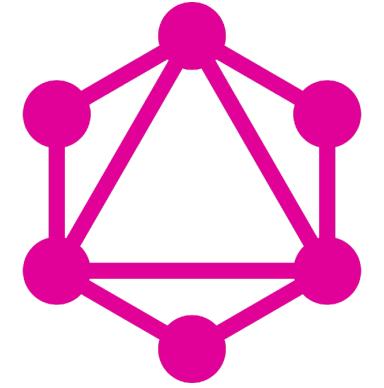
- Schéma : Schema-first VS Code-first
- Relations 1-N et N-N
- Types personnalisés (Enumération / Union / Interface)
- Fragments
- Continuons notre gestionnaire d'évènement

Petites parenthèses



<https://spec.graphql.org/September2025/>

Schema-first VS Code-first



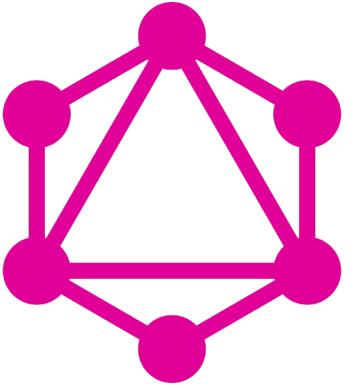
Schema-first

- On définit le schéma avant même d'écrire les revolvers
- Le schéma sert de contrat fort et peut être lu par toute l'équipe, indépendamment du langage utilisé

Code-first

- Le schéma est généré à partir du code (classe ou décorateur en fonction du langage utilisé)
- Evite la duplication des définitions, mais rends le schéma moins lisible

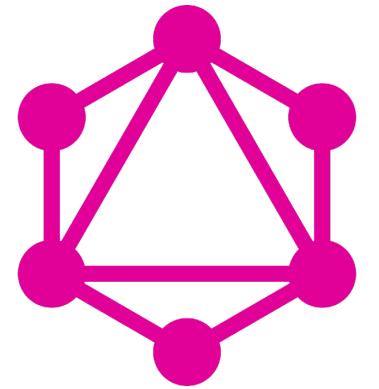
Schema-first VS Code-first



Exemples

Schema-first

```
1 type User {  
2   id: ID!  
3   name: String!  
4 }  
5  
6 type Query {  
7   users: [User!]!  
8 }
```



Schema-first VS Code-first

Exemples

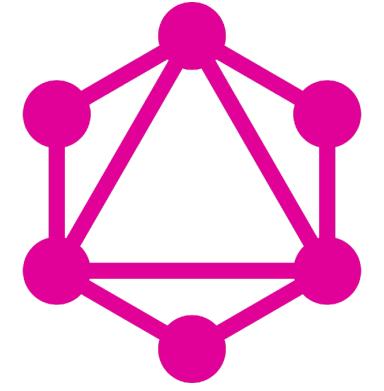
Code-first (TypeScript - NestJS)

```
1 @ObjectType()
2 class User {
3   @Field(() => ID)
4   id: string;
5
6   @Field()
7   name: string;
8 }
9
10 @Resolver(() => User)
11 class UserResolver {
12   @Query(() => [User])
13   users(){
14     return [
15       id: "1",
16       name: "Alice"
17     ]
18   }
19 }
```

Code-first (Python)

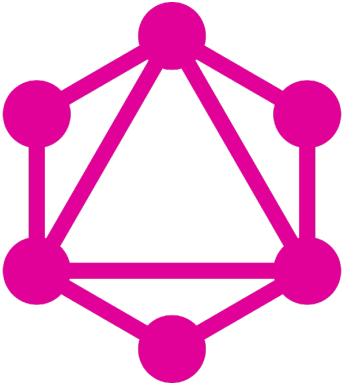
```
1 import graphene
2
3 class User(graphene.ObjectType):
4   id = graphene.ID(required=True)
5   name = graphene.String(required=True)
6
7 class Query(graphene.ObjectType):
8   users = graphene.List(User)
9
10 def resolve_users(root, info):
11   return [User(id="1", name="Alice")]
12
13 schema = graphene.Schema(query=Query)
```

Relation 1-N et N-N



- Les relations sont représentées par des champs dont le type est un autre objet.
- Une **relation 1-N** est modélisée par **un champ de type liste sur la cible**.
- Une **relation N-N** implique **un champ liste des deux côtés**.
- Chaque champ doit avoir un `resolver` qui renvoie une valeur du type annoncé dans le schéma

Relation 1-N et N-N



Exemples

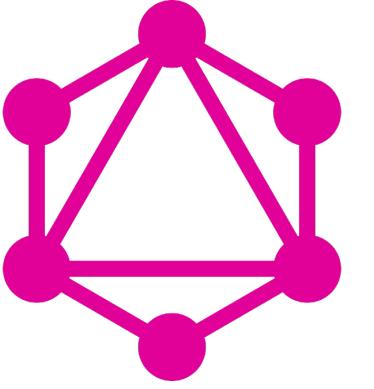
Relation 1-N

```
1 type User {  
2   id: ID!  
3   name: String!  
4   events: [Event!]  
5 }  
6  
7 type Event {  
8   id: ID!  
9   title: String!  
10  organizer: User!  
11 }
```

Relation N-N

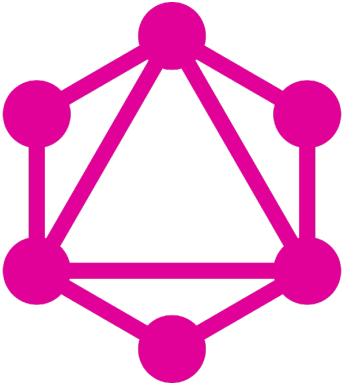
```
1 type User {  
2   id: ID!  
3   name: String!  
4   attending: [Event!]  
5 }  
6  
7 type Event {  
8   id: ID!  
9   title: String!  
10  participants: [User!]  
11 }
```

Types personnalisés



- Les types personnalisés enrichissent votre schéma et reflètent mieux votre domaine. Les **Scalars** et **Enums** constituent les feuilles des arbres de réponses, tandis que les objets forment les noeuds intermédiaires.
- GraphQL prends en charge deux types abstraits : **interfaces** et **unions**.
- Les **enums** restreignent un champs à une liste de valeurs possibles
- Les **interfaces** définissent des champs communs que plusieurs types **doivent** implémenter
- Les **unions** permettent à un champ de renvoyer plusieurs types (cas alternatifs)

Types personnalisés



Exemples

Enum

```
1 enum Role {  
2   ADMIN  
3   STUDENT  
4   TEACHER  
5 }  
6  
7 type User {  
8   id: ID!  
9   name: String!  
10  role: Role!  
11 }
```

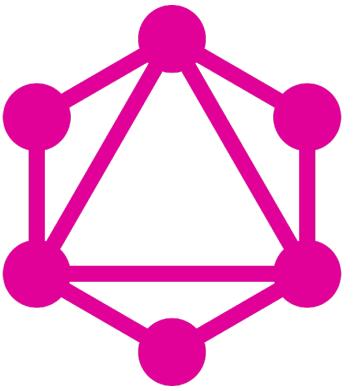
Interface

```
1 interface Person {  
2   id: ID!  
3   name: String!  
4 }  
5  
6 type User implements Person {  
7   id: ID!  
8   name: String!  
9   role: String  
10 }  
11  
12 type Organizer implements Person {  
13   id: ID!  
14   name: String!  
15   company: String  
16 }
```

Unions

```
1 union SearchResult = User | Event  
2  
3 type Query {  
4   search(text: String!): [SearchResult!]!  
5 }
```

Fragments

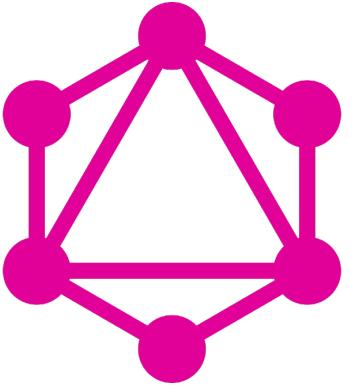


- Vous connaissez le DRY (Don't Repeat Yourself) ?
- Si on répète tous les mêmes champs, il y a possibilité de “factoriser” nos champs pour les appliquer plusieurs fois

Fragments

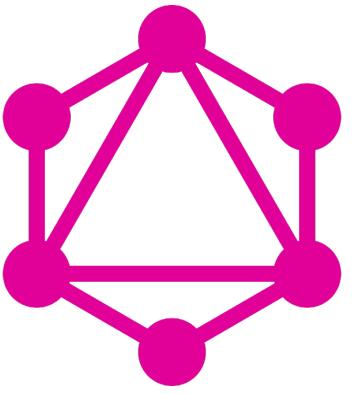
```
1 fragment UserInfo on User {  
2   id  
3   name  
4 }  
5  
6 query {  
7   event(id: "2") {  
8     title  
9     organizer {  
10       ...UserInfo  
11     }  
12   }  
13   users {  
14     ...UserInfo  
15   }  
16 }
```

Un projet

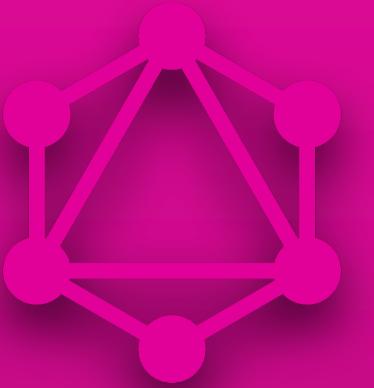


- Continuons notre projet avec les consignes suivantes :
 - Ajouter une relation participants au type Event.
 - Définir un type DateRange pour encapsuler la période d'un événement.
 - Utiliser un fragment UserInfo pour factoriser les champs id et name.

Un projet

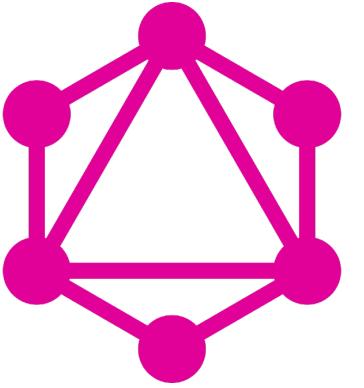


- <https://github.com/othila-academy/graphql-course>
- Des petites choses à challenger...



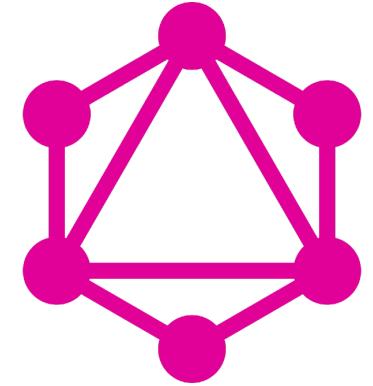
Séance 3 : Mutation, sécurité et clients

De quoi allons-nous parler ?



- Mutations, Arguments et Variables
- Authentification
- Menaces et protections
- Directives
- Client GraphQL
- Continuons notre gestionnaire d'évènement

Mutations, Arguments et Variables

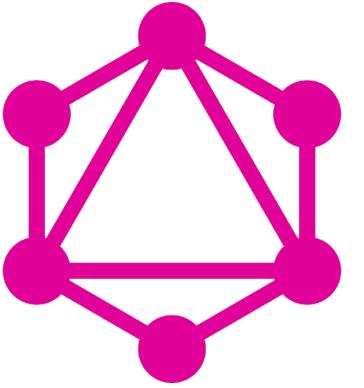


Rappel

- Les **Query** permettent d'accéder à la donnée en lecture. C'est l'équivalent du **GET** en REST.
- Les **Mutation** permettent d'accéder à la donnée en écriture. C'est l'équivalent des verbes **POST / PATCH / PUT / DELETE** en REST.
- Dans le schéma, cela se traduit par un champ de type **Mutation**.
Par exemple :

```
1 type Mutation {  
2   createEvent(input: CreateEventInput!): Event!  
3   joinEvent(eventId: ID!): Event!  
4   deleteEvent(id: ID!): Boolean  
5 }
```

Mutations, Arguments et Variables

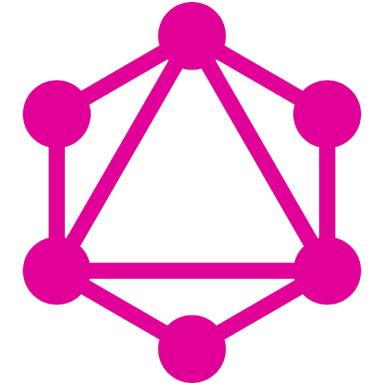


- À noter qu'à la différence des **types** pour les **Query**, il faudra maintenant définir des **input** pour les **Mutation**.

```
1 type Mutation {  
2   createEvent(input: CreateEventInput!): Event!  
3   joinEvent(eventId: ID!): Event!  
4   deleteEvent(id: ID!): Boolean  
5 }
```

```
1 input CreateEventInput {  
2   title: String!  
3   date: String!  
4   description: String  
5 }
```

Mutations, Arguments et Variables



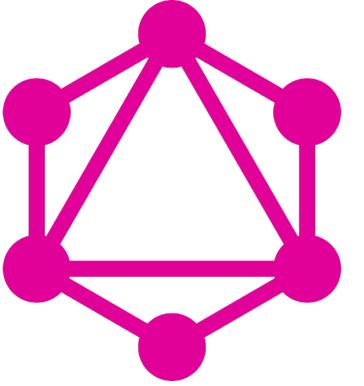
Mutations

- Les **Mutation** renvoient l'objet modifié pour synchroniser le client. Et respecte donc les mêmes règles pour les Query. On ne renvoie que les informations nécessaires.

Mutation

```
1 mutation{
2   createEvent(title: "Soirée étudiante", date: "2025-10-15") {
3     id
4     title
5     date
6   }
7 }
```

Mutations, Arguments et Variables



Arguments et variables

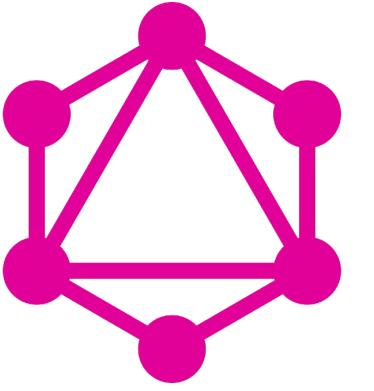
- Les champs et **Mutation** peuvent recevoir des arguments. Les variables, quant à elle, permettent de réutiliser une requête avec des valeurs dynamiques.

Exemple avec variables

```
1 mutation CreateEvent($title: String!, $date: String!){  
2   createEvent(title: $title, date: $date) {  
3     id  
4     title  
5     date  
6   }  
7 }
```

```
1 {  
2   "title": "Conférence",  
3   "date": "2025-11-01"  
4 }
```

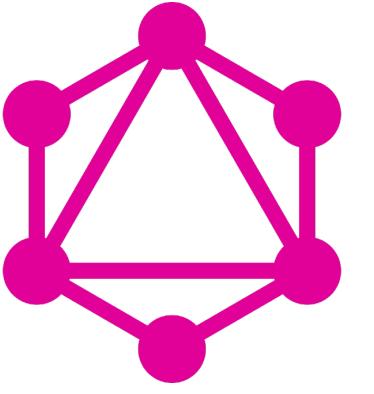
Authentification



- GraphQL n'a pas de mécanisme intégré
- Usage courant : **JWT** (Json Web Token)
 - Le token est envoyé dans les header HTTP
 - Vérification dans les resolvers selon le rôle utilisateur

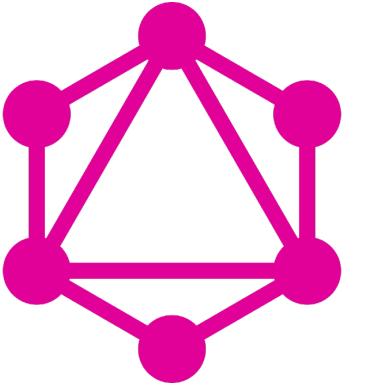
```
1 Authorization: Bearer <token>
```

Authentification



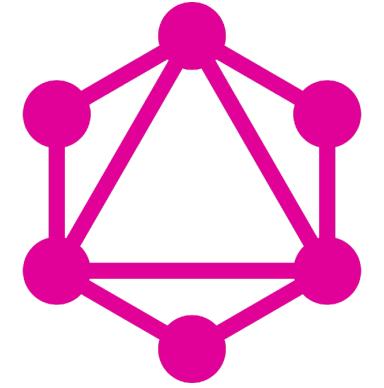
- Le **contexte GraphQL** est alors **enrichi** de l'utilisateur authentifié, de ses rôles, permissions, etc.
- Dans les **resolvers** de **Mutation** (et de **Query**), on vérifie si l'utilisateur a le droit d'exécuter l'action.

Menaces et protections



- La construction du schéma et la manière dont notre authentification est gérée peuvent avoir un **impact significatif** sur la sécurité de notre serveur.
- Voici quelques menaces connues et comment les éviter :

Menaces et protections



Menaces

Requêtes trop profondes (Deep nesting)

Injections / manipulations d'arguments

Requêtes larges (Breadth)

Exposition d'erreurs sensibles

Introspection abusive

DoS

Protections

→ Imposer une limite de profondeur (maxDepth)

→ Valider/assainir les inputs, utiliser des requêtes paramètres

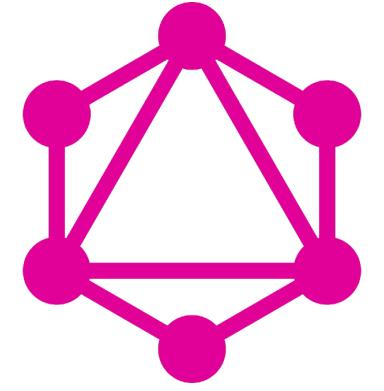
→ Limiter le nombre de champs, paginer les listes

→ Master les détails d'erreur, log interne, retourner des erreurs génériques

→ Désactiver l'introspection en production ou la restreindre selon les rôles

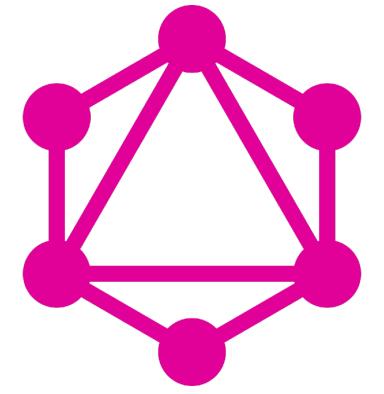
→ Limiter le taux (rate limiting), timeouts, quotas

Directives



- Le **contexte GraphQL** avec le token JWT nous permet d'autoriser ou non une action à l'utilisateur. Mais nous pouvons **affiner** ces autorisations d'accès au niveau des champs du schéma et déléguer la vérification d'accès à un middleware.
- Pour cela, nous allons dans le schéma inclure des mots-clés **préfixés par un “@”**. La spécification nous donne une liste de directive pré-existante : **@skip, @include, @deprecated, @specifiedBy, @oneOf**. Mais **vous pouvez tout à faire créer les votre !**
- Dans le cas de création de directive custom, il faut **s'assurer que les implémentations de serveur supportent cette fonctionnalité**. Par exemple Apollo Server nécessite des librairies comme **@graphql-tools** pour appliquer le comportement.

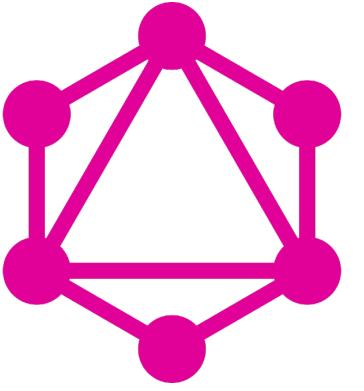
Directives



```
1 query ($withDate: Boolean!) {  
2   events {  
3     title  
4     date @include(if: $withDate)  
5   }  
6 }
```

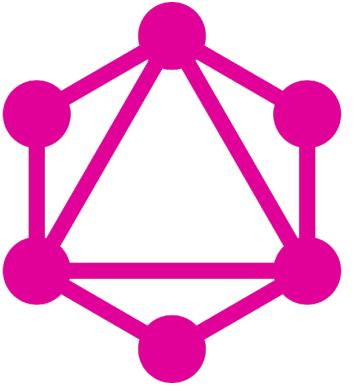
```
1 directive @auth(  
2   requires: Role = ADMIN  
3 ) on FIELD_DEFINITION | OBJECT  
4  
5 type Mutation {  
6   createEvent(input: CreateEventInput!): Event! @auth(requires: ORGANISATEUR)  
7 }
```

Client GraphQL

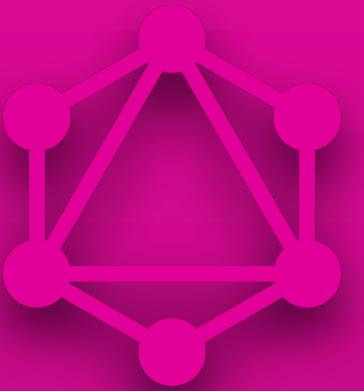


- Une fois le serveur **GraphQL** sécurisé et l'ensemble de nos **Query/Mutation** prêtes, le client doit les consommer, gérer son cache, etc.
- Vous avez déjà utilisé un client (plus léger) **GraphQL : Apollo Sandbox**
- Mais un vrai client (Application React par exemple) doit être en capacité de gérer plus de cas :
 - Envoie des opérations vers le serveur
 - Peut utiliser un cache local pour minimiser un appel réseau
 - Gérer les erreurs, états de chargement, réponses optimistes
 - Peut combiner les données locales et distantes

Un projet

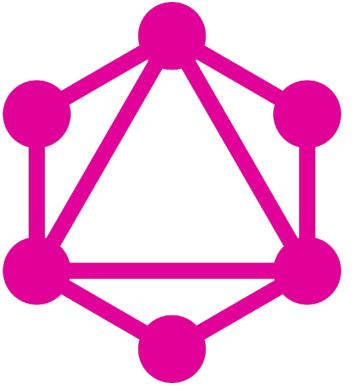


- <https://github.com/othila-academy/graphql-course>
- Découpage sur une heure :
 - Full API : CRUD complet sur les User et Event (au minimum)
 - Intégration de l'authentification (JWT)
 - PAUSE
 - Intégration FRONT



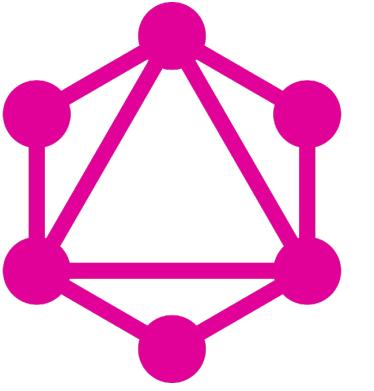
Séance 4 : Performance et temps réel

De quoi allons-nous parler ?

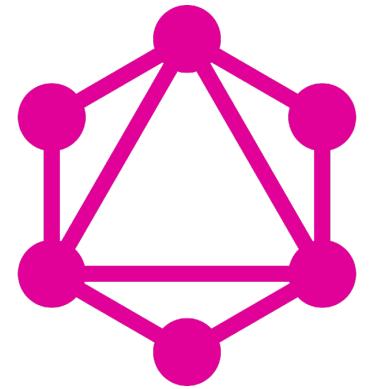


- Pagination des requêtes : 2 méthodes Offset/limit et Cursor based
- Subscriptions
- Persisted Queries
- Retour sur notre projet

Pagination des requêtes



- Pour rappel un champ qui renvoie un grand nombre d'objets **doit** proposer des mécanismes de paginations.
- Plusieurs stratégies existent : offset/limit et cursor-based



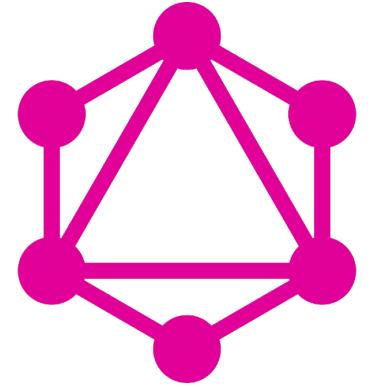
Pagination des requêtes

Offset/limit

- Principe : deux arguments sur la **Query** : **limit** (ou **first**) pour le nombre maximum d'éléments et **offset** (ou **skip**) pour indiquer combien d'éléments ignorer.

```
1 type Query {  
2   signedUpUsers(limit: Int, offset:Int): [Users!]!  
3 }
```

```
1 type Query {  
2   signedUpUsers(limit: Int, offset:Int): [Users!]!  
3 }
```

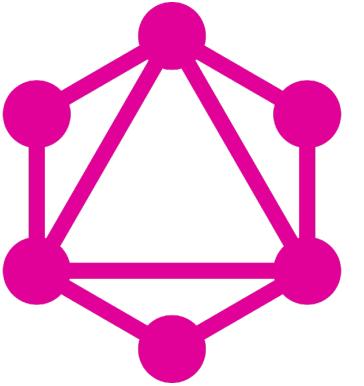


Pagination des requêtes

Offset/limit

- **Avantages :**
 - Très simple à mettre en oeuvre
 - bien supporté par les bases SQL (via LIMIT/OFFSET) et NOSQL
- **Inconvénients :**
 - Les changements dans les données entre deux requêtes peuvent provoquer des doublons ou des éléments manquants
 - Impossible d'obtenir directement la dernière page ni de savoir s'il reste des pages à récupérer.
 - Absence d'informations comme totalCount, hasNextPage ou hasPreviousPage
- **Quand utiliser ?** Listes peu dynamiques, interfaces simples, ou lorsque la duplication n'est pas critique

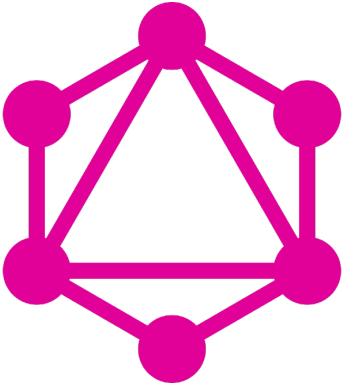
Pagination des requêtes



Cursor based

- Au lieu d'utiliser un *offset*, on utilise un *curseur opaque* (souvent encodé en base64). Chaque noeud est enveloppé dans un *Edge* qui contient le *cursor* et la node *réelle*.
- Ce type de pagination utilise des arguments **first + after** pour paginer vers **l'avant** et **last+before** pour paginer vers **l'arrière**.
- C'est la méthode la plus fiable ! *Mais peut être moins facile à mettre en place !*
- Quelques définitions s'imposent...

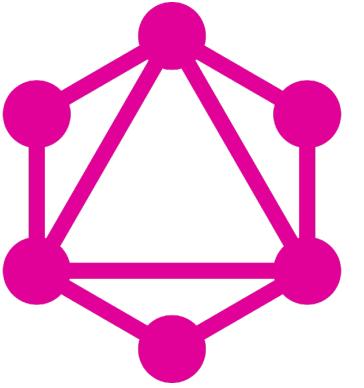
Pagination des requêtes



Cursor based

- edges : tableau contenant les objets (chaque liaison représente un résultat)
- node : l'élément réel attendu (ex : un post, avec ses champs ID, title, etc.)
- cursor : une valeur opaque qui permet de récupérer la page suivante
- pageInfo : informations supplémentaires
(exemple : hasNextPage, endCursor, hasPreviousPage, etc.)

Pagination des requêtes



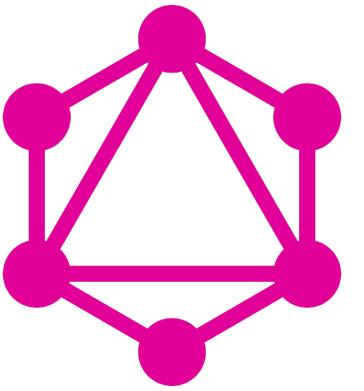
Cursor based

- **Avantages :**

- Stable lorsque les données évoluent rapidement (pas de duplication)
- Permet la pagination bidirectionnelle et l'indication de fin de liste
- Donne plus de métadonnées (totalCount, startCursor, endCursor) pour améliorer l'UX

- **Inconvénients :**

- Plus verbeux et plus complexe à implémenter
- Impossible de “sauter directement” à une page arbitraire sans parcourir les pages précédentes (il faut récupérer les curseurs)
- **Quand utiliser ?** Préférable pour des flux dynamiques (fil d'actualité, chat), quand on veut éviter les doublons et offrir une expérience utilisateur riche



Pagination des requêtes

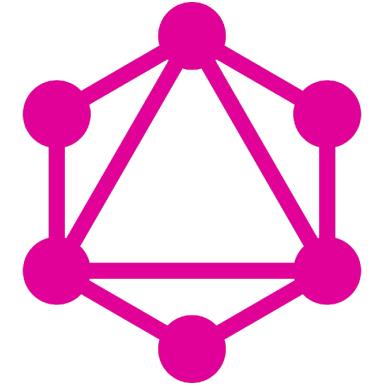
Cursor based

Exemple...

```
1 query {  
2   signedUpUsers(first: 10, after: "<cursor>") {  
3     edges {  
4       node { id name }  
5       cursor  
6     }  
7     pageInfo {  
8       endCursor  
9       hasNextPage  
10    }  
11    totalCount  
12  }  
13 }
```

- Une démo s'impose...

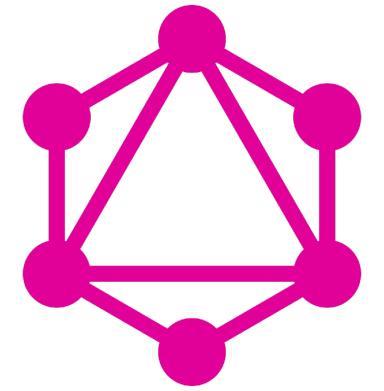
Subscriptions



Rappel

- Il existe 3 types d'opération en GraphQL : Les **Query**, les **Mutation** et les **Subscription**.
- Les **Query** permettent d'accéder à la donnée en lecture. C'est l'équivalent du **GET** en REST.
- Les **Mutation** permettent d'accéder à la donnée en écriture. C'est l'équivalent des verbes **POST / PATCH / PUT / DELETE** en REST.
- Les **Subscription** va permettre de recevoir des mises à jour **en temps réel**.

Subscriptions

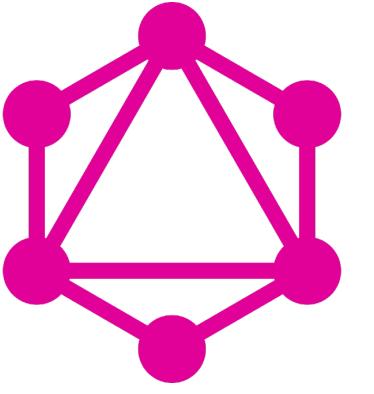


- Une **Subscription** est une requête longue durée : C'est à dire que le serveur diffuse des messages au client lorsque des événements se produisent (nouveau message, nouvel inscription...)
- Bien que les **Query** et **Mutation** passent par le protocole **HTTP**, les **Subscription** passe généralement par le protocole **WebSocket** (ou bien le **Server-Sent Event** si le système le permet). Elles nécessitent souvent un système pub/sub pour publier et distribuer les événements aux abonnés.



À noter que le protocole n'est pas spécifié dans la spécification GraphQL.

Subscriptions

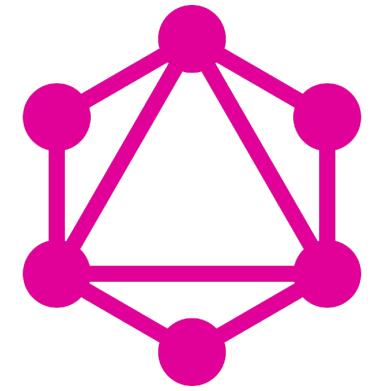


- Dans le schéma, cela se traduit par un champ de type **Subscription**.

Par exemple :

```
1 type Subscription {  
2   reviewCreated: Review  
3   commentAdded: Comment  
4 }
```

Subscriptions



- D'un point de vue requête client, ce sont quasiment mêmes règles que pour les **Query**.
Pratique ! Cependant, il y a quelques règles qu'il faut respecter :
- **Un seul champ racine** uniquement. Contrairement aux **Query**, on ne peut pas cumuler les requêtes au sein d'une même suscription. Il faudra en faire une à chaque fois.

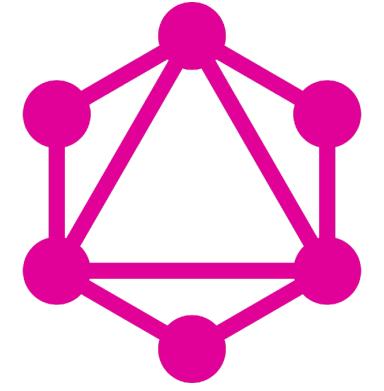
```
1 subscription {  
2   reviewCreated {  
3     rating  
4     commentary  
5   }  
6   humanFriendsUpdated {  
7     name  
8     friends {  
9       name  
10    }  
11  }  
12 }
```



```
1 subscription NewReviewCreated {  
2   reviewCreated {  
3     rating  
4     commentary  
5   }  
6 }  
7 subscription FriendListUpdated($id: ID!) {  
8   humanFriendsUpdated(id: $id) {  
9     name  
10    friends {  
11      name  
12    }  
13  }  
14 }
```

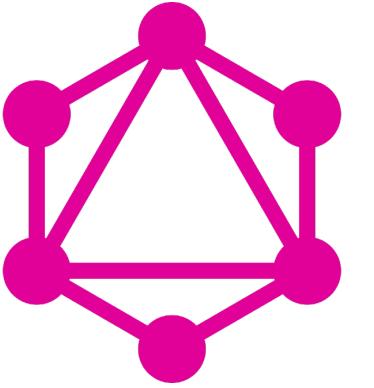


Subscriptions



- D'un point de vue requête client, ce sont quasiment mêmes règles que pour les **Query**.
Pratique ! Cependant, il y a quelques règles qu'il faut respecter :
 - Les directions **@skip** et **@include** sont **interdites** au niveau racine
 - Le champ racine ne doit pas être un champ **d'introspection** (ex: `__typename`)
 - Une démo s'impose...

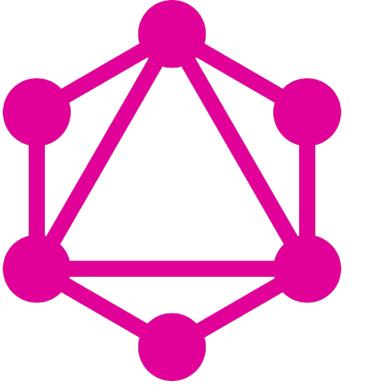
Persisted Queries & cache



Persisted Queries

- **Le problème :** En **GraphQL**, chaque requête (**Query/Mutation**) est envoyé sous forme de texte au serveur. Ces requêtes peuvent être longues, donc :
 - Elles *prennent de la place* dans le corps HTTP
 - Elles ne peuvent *pas toujours être mises en cache* facilement,
 - Elles peuvent *exposer des informations sensibles* (le texte de la requête)
 - Et elles *ralentissent les performances* (validation à chaque envoi).

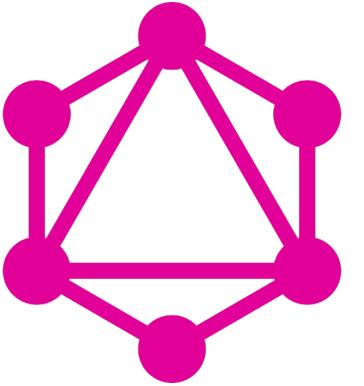
Persisted Queries & cache



Persisted Queries

- **La solution :** Les requêtes persistées.
- On enregistre à l'avance les requêtes côté serveur et on n'envoie plus le texte complet.
- Le client envoie seulement un identifiant (hash) de la requête.

Persisted Queries & cache



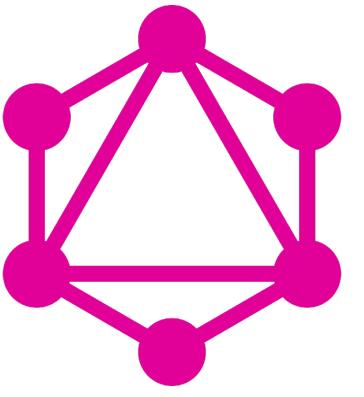
Persisted Queries

```
1 // Côté client
2 {
3   "id": "hash_1234abcd",
4   "variables": { "userId": "1" }
5 }
```

Traduis par
le serveur

```
1 query getUser($userId: ID!) {
2   user(id: $userId) {
3     firstname
4     lastname
5   }
6 }
```

Un projet



- <https://github.com/othila-academy/graphql-course>
- Regardons où vous en êtes...