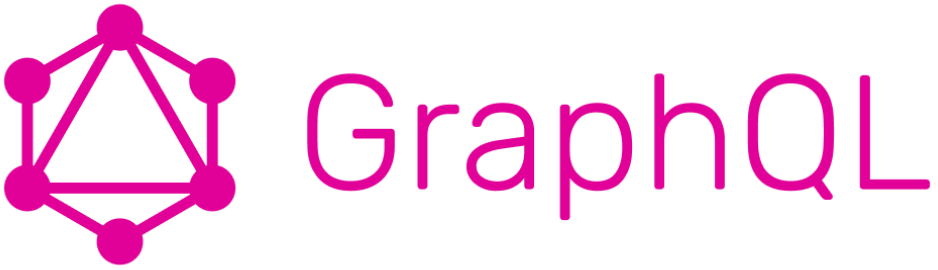

GraphQL

Table des matières

Historique de GraphQL

Origine et évolution

GraphQL trouve ses racines au sein de Facebook, où il a été développé en 2012 par une équipe dirigée par Lee Byron, Dan Schafer et Nick Schrock. À l'époque, Facebook cherchait une solution pour optimiser la récupération de données dans leurs applications mobiles, confrontées à des défis spécifiques liés à la variété des informations nécessaires et à la limitation des ressources mobiles.

Le besoin de créer une API plus flexible et efficiente a conduit à la conception de GraphQL, un langage de requête permettant aux clients de spécifier précisément les données dont ils avaient besoin. Après des années de développement en interne, Facebook a dévoilé GraphQL au public en 2015 en le libérant en tant que projet open source.

L'impact rapide de GraphQL a été renforcé par son adoption croissante au sein de la communauté des développeurs. Des entreprises telles que GitHub, Shopify et Twitter ont rapidement reconnu les avantages de cette technologie et ont contribué à son développement continu. Aujourd'hui, GraphQL est bien plus qu'une solution interne de Facebook ; c'est devenu un standard de facto pour la construction d'API flexible et performante, soutenu par une communauté mondiale de développeurs dévoués.

Comprendre GraphQL

Fondements conceptuels

GraphQL, au cœur de son architecture, représente une innovation significative dans le domaine des API en offrant une approche radicalement différente de la récupération de données par rapport aux méthodes traditionnelles.

Au lieu de suivre le modèle rigide des API REST, où les clients sont souvent confrontés à des données préalablement définies par le serveur, GraphQL propose une approche plus flexible. Il se positionne comme un langage de requête permettant aux clients de spécifier les données qu'ils souhaitent obtenir, et ce, de manière granulaire. Cette approche s'oppose au transfert d'informations inutiles, communément appelé "surchargement" de données, qui peut survenir avec les méthodes REST, où le client reçoit souvent plus d'informations qu'il n'en a réellement besoin.

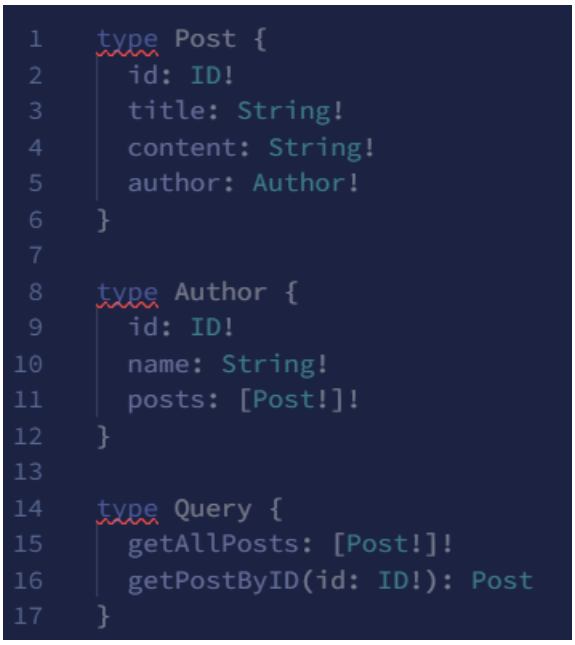
Le modèle de données de GraphQL est basé sur un graphe, ce qui signifie que les relations entre les différentes entités sont représentées de manière visuelle et structurelle. Chaque entité est un nœud, et les relations entre ces entités sont les arêtes du graphe. Cette représentation sous forme de graphe offre une vue intuitive des données et permet aux clients de naviguer efficacement entre les différentes parties du système.

Schéma GraphQL

Structure du schéma

Le schéma GraphQL représente l'épine dorsale de toute implémentation GraphQL, agissant comme un contrat explicite entre le serveur et le client. Il définit la structure des données disponibles, spécifiant les types d'objets, les relations entre eux, et les opérations permises. La clarté du schéma est cruciale, car elle facilite la compréhension commune entre les équipes frontend et backend, établissant ainsi une base solide pour le développement collaboratif.

Un exemple simple de schéma pourrait être celui d'une application de blog, comprenant des types tels que *Post* et *Author*. Le schéma indiquerait comment ces types sont liés, quels champs sont disponibles sur chaque type, et quelles opérations peuvent être effectuées. Par exemple :



```graphql
type Post {
  id: ID!
  title: String!
  content: String!
  author: Author!
}

type Author {
  id: ID!
  name: String!
  posts: [Post!]!
}

type Query {
  getAllPosts: [Post!]!
  getPostByID(id: ID!): Post
}
```

Ici, le schéma spécifie que chaque *Post* a un identifiant *(id)*, un titre *(title)*, un contenu *(content)*, et est associé à un auteur *(author)*. De même, chaque *Author* a un identifiant, un nom, et peut avoir plusieurs articles *(posts)*. La section Query définit les opérations que le client peut effectuer, telles que récupérer tous les articles ou obtenir un article par son identifiant.

Types de données

GraphQL supporte une gamme étendue de types de données, permettant aux développeurs de modéliser de manière précise la structure des informations. Ces types incluent les scalaires (comme *String*, *Int*, *Float*, *Boolean*, et *ID*), les objets (décrivant des structures complexes avec plusieurs champs), les listes (collections d'éléments du même type), et les interfaces (définissant des ensembles de champs communs à plusieurs types).

Prenons l'exemple d'un type scalaire et d'un type objet :

```
type Person {
  id: ID!
  name: String!
  age: Int!
}

type Query {
  getPersonByID(id: ID!): Person
}

```

Ici, *Person* est un type objet représentant une personne avec un identifiant, un nom, et un âge. Le schéma permet au client de récupérer une personne par son identifiant avec la requête *getPersonByID*.

Opérations GraphQL

Query

Les opérations de requête (Query) constituent le moyen par excellence pour les clients de demander des données spécifiques au serveur GraphQL. Contrairement aux requêtes traditionnelles où le serveur dicte la structure des données renvoyées, la requête GraphQL permet au client de spécifier précisément les champs et les relations nécessaires. Cela se traduit par des réponses plus ciblées, minimisant le transfert d'informations non pertinentes sur le réseau.

Un exemple concret de requête GraphQL pour obtenir des informations sur un utilisateur pourrait ressembler à ceci :

```
query {
  getUserByID(id: "123") {
    id
    name
    email
    posts {
      title
      content
    }
  }
}

```

Ici, la requête demande au serveur de renvoyer l'identifiant, le nom, et l'e-mail de l'utilisateur avec l'ID "123", ainsi que les titres et le contenu de tous les articles associés à cet utilisateur.

Mutation

Les mutations sont utilisées lorsque des modifications doivent être apportées aux données côté serveur. Elles offrent une manipulation complète des données, permettant d'ajouter de nouveaux éléments, de mettre à jour des informations existantes, ou même de supprimer des données. Les mutations garantissent une interaction bidirectionnelle entre le client et le serveur pour assurer la cohérence des données.

Un exemple de mutation GraphQL pour ajouter un nouvel utilisateur pourrait ressembler à ceci :

Cette mutation crée un nouvel utilisateur avec le nom "John Doe" et l'e-mail *john@example.com* et renvoie l'identifiant, le nom, et l'e-mail du nouvel utilisateur créé.

Subscription

Les abonnements (Subscriptions) introduisent la dimension de la communication en temps réel dans GraphQL. Ils permettent au serveur d'envoyer des mises à jour instantanées au client dès que des changements surviennent du côté serveur. Les abonnements sont particulièrement utiles pour les fonctionnalités nécessitant une réactivité en temps réel, comme les flux d'actualités ou les systèmes de chat.

Un exemple de souscription GraphQL pour être informé en temps réel des nouveaux messages dans un système de chat pourrait ressembler à ceci :

Cette souscription permet au client de recevoir instantanément les nouveaux messages avec les informations sur l'expéditeur et le contenu dès qu'ils sont ajoutés côté serveur.

Avantages de GraphQL

Comparaison avec d'autres solutions API

Lorsqu'on compare GraphQL avec les solutions API REST traditionnelles, plusieurs avantages significatifs émergent, mettant en évidence la flexibilité et l'efficacité de GraphQL.

- Souplesse dans la récupération des données :

GraphQL permet aux clients de spécifier précisément les données dont ils ont besoin. Contrairement aux API REST où les données sont prédéfinies par le serveur, GraphQL offre une granularité fine dans la récupération des informations. Les clients peuvent demander uniquement les champs nécessaires, éliminant ainsi le surchargement d'informations inutiles.

- Réduction du nombre de requêtes réseau :

En utilisant GraphQL, un client peut regrouper plusieurs requêtes en une seule. Cela évite la surcharge liée à l'obtention d'informations par le biais de multiples appels réseau, comme c'est souvent le cas avec les API REST. Une seule requête GraphQL peut être formulée pour récupérer toutes les données nécessaires, ce qui améliore considérablement les performances en réduisant la latence et la charge sur le réseau.

- Évolutivité du schéma :

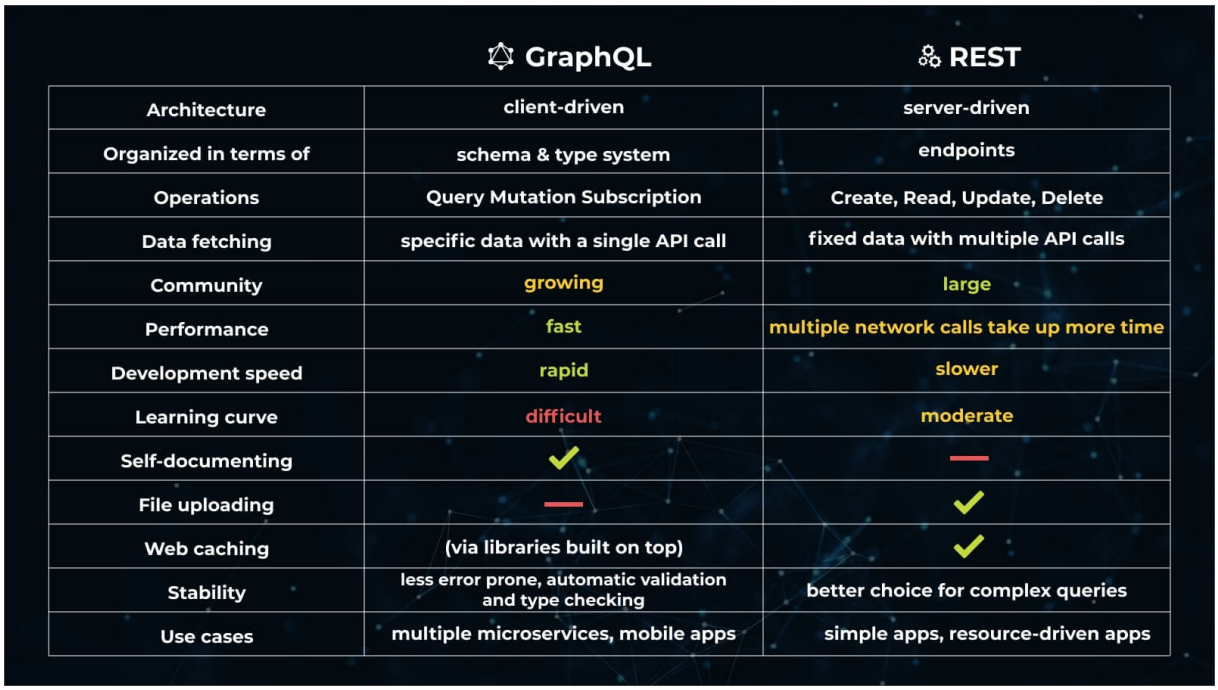
GraphQL facilite l'évolution du schéma sans impacter les clients existants. Les nouvelles fonctionnalités peuvent être ajoutées au schéma sans créer de versions multiples de l'API, ce qui simplifie la maintenance et la gestion des API à long terme.

- Documentation auto-générée :

GraphQL génère automatiquement une documentation complète et interactive de l'API, basée sur le schéma. Cela simplifie la compréhension de l'API par les développeurs, encourageant ainsi l'adoption rapide et la collaboration entre les équipes frontend et backend.

- Optimisation des performances côté client :

Avec GraphQL, les clients peuvent éviter de récupérer des données redondantes, réduisant ainsi la charge de travail côté client. Cette capacité à définir précisément les données nécessaires améliore l'efficacité des applications, surtout dans des contextes où la bande passante et les ressources client sont limitées.



	GraphQL	REST
Architecture	client-driven	server-driven
Organized in terms of	schema & type system	endpoints
Operations	Query Mutation Subscription	Create, Read, Update, Delete
Data fetching	specific data with a single API call	fixed data with multiple API calls
Community	growing	large
Performance	fast	multiple network calls take up more time
Development speed	rapid	slower
Learning curve	difficult	moderate
Self-documenting	✓	—
File uploading	—	✓
Web caching	(via libraries built on top)	✓
Stability	less error prone, automatic validation and type checking	better choice for complex queries
Use cases	multiple microservices, mobile apps	simple apps, resource-driven apps

[1]

Création d'un Schéma GraphQL

Définition des types

La première étape cruciale dans la création d'un schéma GraphQL consiste à définir les types de données nécessaires à l'application. Ces types représentent les entités fondamentales avec lesquelles l'application interagira. Chaque type est composé de champs, définissant les propriétés spécifiques de cette entité.

Par exemple, dans le contexte d'une application de réseau social, les types peuvent inclure *User*, *Post*, et *Comment*. La définition du type *User* pourrait ressembler à ceci :

```
type User {
  id: ID!
  username: String!
  email: String!
  posts: [Post!]!
}
```

Ici, le type *User* a des champs tels que l'identifiant *(id)*, le nom d'utilisateur *(username)*, l'e-mail *(email)*, et une liste de messages *(posts)* associés à cet utilisateur.

[1] https://www.mobilelive.ca/blog/graphql-vs-rest-what-you-didnt-know

Relations entre types

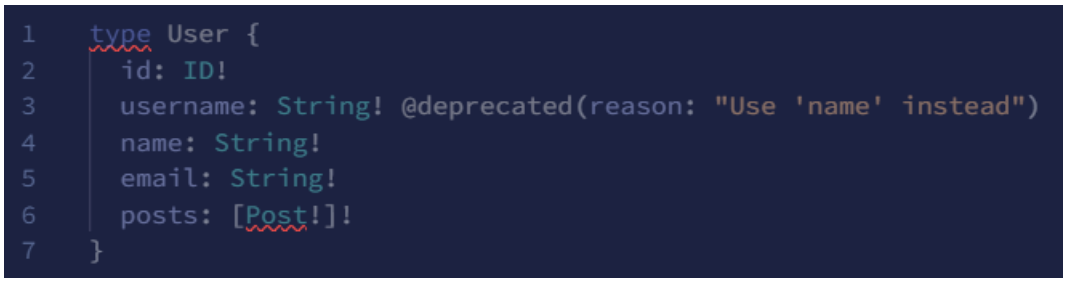
Les relations entre types définissent la manière dont les données sont liées les unes aux autres. Cette étape est cruciale pour modéliser des structures complexes dans l'application. Reprenons l'exemple de l'application de réseau social et ajoutons une relation entre les types *User* et *Post* :

Ici, le champ *author* du type *Post* est lié au type *User*, indiquant que chaque message a un auteur qui est un utilisateur.

Directives GraphQL

Les directives fournissent un mécanisme puissant pour modifier le comportement du schéma GraphQL. Elles offrent une flexibilité supplémentaire lors de la définition des types et des champs, permettant aux développeurs de personnaliser le comportement de l'API en fonction des besoins spécifiques.

Par exemple, la directive *@deprecated* peut être utilisée pour marquer un champ comme obsolète, fournissant des informations utiles aux développeurs utilisant l'API :



Ici, le champ *username* est marqué comme obsolète avec un message expliquant la raison du changement, aidant ainsi les développeurs à migrer vers l'utilisation du champ *name* à la place.

Donc la création d'un schéma GraphQL est un processus itératif où les types, les relations entre types et les directives sont définis de manière à représenter précisément la structure des données tout en offrant une flexibilité d'évolution et de personnalisation.

Sécurité dans GraphQL

Considérations de sécurité

Bien que GraphQL offre une flexibilité considérable dans la récupération de données, cela s'accompagne de responsabilités accrues en matière de sécurité. Les développeurs doivent être attentifs à plusieurs aspects pour assurer un environnement sécurisé.

- Authentification :

L'authentification est cruciale pour garantir que seules les personnes autorisées peuvent accéder aux ressources protégées. Les mécanismes d'authentification standard, tels que l'utilisation de jetons d'accès (tokens) ou de flux OAuth, doivent être mis en place. GraphQL s'intègre souvent harmonieusement avec ces méthodes pour sécuriser les points d'entrée de l'API.

- Autorisation :

Une fois qu'un utilisateur est authentifié, l'autorisation détermine quelles opérations sont permises. GraphQL propose des mécanismes d'autorisation granulaires, permettant de spécifier quel utilisateur a accès à quelles parties du schéma. Les développeurs doivent s'assurer que seuls les utilisateurs autorisés peuvent effectuer certaines requêtes ou mutations.

- Validation des requêtes :

La validation des requêtes est un élément clé de la sécurité GraphQL. Il est essentiel de mettre en place des mécanismes pour valider les requêtes entrantes, en s'assurant qu'elles respectent les règles définies. Cela peut inclure la limitation de la profondeur des requêtes, la vérification des champs demandés, et la prévention des requêtes trop complexes susceptibles de causer des dénis de service (DoS).

- Protection contre les injections de données :

Les injections de données, bien que moins courantes que dans d'autres types d'API, peuvent toujours être une menace potentielle. Les entrées utilisateur doivent être validées et traitées avec soin pour éviter toute injection de code malveillant. L'utilisation de bibliothèques de sécurité et la validation rigoureuse des entrées peuvent aider à atténuer ce risque.

- Audit des requêtes :

La mise en place de mécanismes d'audit pour enregistrer et suivre les requêtes GraphQL peut être cruciale pour la détection précoce de comportements suspects ou d'activités malveillantes. Les journaux d'audit aident les équipes de sécurité à surveiller l'utilisation de l'API et à réagir rapidement en cas d'incident.

GraphQL Playground

https://legacy.graphqlbin.com/new

https://codesandbox.io/p/sandbox/node-apollo-graphql-forked-ndrnkd

Bibliographie

graphql_tutorial. (n.d.). Retrieved from https://www.tutorialspoint.com/: https://www.tutorialspoint.com/graphql/graphql_tutorial.pdf

GrapQL. (n.d.). *learn*. Retrieved from https://graphql.org/: https://graphql.org/learn/

Pavel_Chertorogov. (n.d.). *the_holy_contract-between_client_and_server.* Retrieved from https://assets.ctfassets.net: https://assets.ctfassets.net/nn534z2fqr9f/5zTslHVxlY6e2wAkMQMceS/d4024a20ab5cea77a9198d5d520ef970/Pavel_Chertorogov_GraphQL_the_holy_contract-between_client_and_server_v11.pdf

Tazetdinov, A. (2023). *Learn how to build scalable and high-performance apps from scratch (English Edition)*. BPB Publications.