Engineer

GraphQL

langage de requêtes où le client choisit les champs

Ce que c’est

GraphQL est un langage de requêtes pour les API, accompagné du moteur qui les exécute sur le serveur. Le client envoie un document désignant exactement les champs voulus et le serveur répond par des données de même forme. La spécification le décrit comme un langage et un moteur d’exécution pour applications client-serveur. Elle indique sa création en 2012, le début du travail de normalisation ouverte en 2015 et, depuis 2019, son statut de livrable du GraphQL Specification Project sous l’égide de la Joint Development Foundation. La dernière version est l’édition de septembre 2025.

Tout part du schéma publié par le service : des types avec des champs nommés, chaque champ ayant son propre type. Les types scalaires intégrés sont `Int`, `Float`, `String`, `Boolean` et `ID` ; un service peut en ajouter, par exemple pour un UUID. Tout type accepte null sauf marquage par `!` : `String!` ne peut jamais être null. Les listes sont écrites entre crochets : `[LineItem!]!` est une liste toujours présente qui ne contient jamais null. Le schéma peut lui-même être interrogé. Avec les champs d’introspection `__schema` et `__type`, un client peut demander ce que propose le service ; outils de développement et bibliothèques clientes reposent sur cette possibilité.

Il existe trois types d’opérations. Une requête est, selon la spécification, « une récupération en lecture seule » ; une mutation est une écriture suivie d’une récupération ; un abonnement est une requête de longue durée fournissant des données en réponse à une succession d’événements au fil du temps. Sur le serveur, chaque champ est associé à une fonction, généralement appelée résolveur, qui produit sa valeur. Le service vérifie d’abord la conformité de l’opération au schéma, puis appelle les résolveurs des champs sélectionnés. Les champs de premier niveau d’une mutation s’exécutent l’un après l’autre, pas en parallèle : deux écritures dans la même requête ne peuvent donc pas entrer en concurrence.

La spécification n'impose ni transport ni format de sérialisation. HTTP est le choix le plus courant : l'API GraphQL est servie à une URL unique, généralement terminée par `/graphql`, et répond en JSON, le format le plus courant. Le client envoie un POST dont le corps JSON porte le texte `query` et, au besoin, ses `variables`. Un serveur peut aussi accepter GET pour les requêtes de lecture, mais une mutation doit toujours être un POST. La réponse contient le résultat sous `data` et les problèmes sous `errors`. Les recommandations de la GraphQL Foundation suivent une spécification GraphQL-over-HTTP encore à l'état de brouillon.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que GraphQL ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. La partie invisible pour les utilisateurs : où résident les données, quelles règles s'appliquent et comment les autres systèmes peuvent y accéder.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Pourquoi c'est important

Dans la plupart des API HTTP, le serveur décide de ce que renvoie chaque point d'accès. Un écran ayant besoin de l'état d'une commande et du nom de son client doit soit recevoir la commande et le client entiers, soit faire une requête par ressource et assembler les réponses. Dans le premier cas, il reçoit plus que demandé. La spécification nomme ces problèmes over-fetching et under-fetching. GraphQL inverse la répartition : le client choisit les champs individuellement et reçoit exactement ceux-là, sans plus. Une requête peut aussi suivre les relations, de la commande à ses lignes puis à son client, parce qu'elle a la forme des données attendues.

Le système de types apporte deux bénéfices. La conformité de chaque opération au schéma est vérifiée avant l'exécution du moindre résolveur : un champ mal orthographié ou un argument du mauvais type fait rejeter toute l'opération avant la moindre exécution. Les outils peuvent repérer ces mêmes erreurs pendant l'écriture du code. Comme le client ne reçoit que les champs choisis, ajouter un champ ne coûte rien aux clients existants. L'API peut ainsi évoluer sans numéro de version : un nouveau champ apparaît à côté de l'ancien, celui-ci reçoit `@deprecated` avec un motif, reste valide pour ne pas casser les clients et disparaît quand plus aucun ne l'utilise.

Les coûts sont réels et viennent surtout du contournement des mécanismes HTTP. Les entités ne sont plus identifiées par des URL : tout arrive à une adresse, généralement en POST, alors que ce sont les requêtes GET qui sont habituellement considérées comme pouvant être mises en cache sans indication explicite. Le cache fourni sans effort à une API REST par les intermédiaires doit être reconstruit délibérément. Le statut ne dit plus tout : une réponse portant des données et des erreurs reste un succès HTTP. Le client compose désormais des requêtes jamais vues du serveur. Une seule peut demander beaucoup de travail ; protéger le serveur de cette charge incombe au service, pas à GraphQL. Les résolveurs par champ sont faciles à écrire, mais une implémentation naïve multiplie les appels vers la base sous-jacente.

À quoi cela ressemble en pratique

Une petite partie d'un schéma de commandes avec lignes et client :

```
type Query {
  order(id: ID!): Order
}

type Order {
  id: ID!
  status: String!
  items: [LineItem!]!
  customer: Customer
}

type LineItem {
  name: String!
  quantity: Int!
}

type Customer {
  name: String!
}
```

Un écran affichant l'état, les lignes et le nom du client envoie cette opération à `/graphql` dans le corps d'un POST, avec `{"id": "7421"}` pour variables :

```
query OrderSummary($id: ID!) {
  order(id: $id) {
    status
    items {
```

```
      name
      quantity
    }
    customer {
      name
    }
  }
}
```

Supposons que la commande et ses lignes soient résolues, mais que le service derrière `customer` soit en panne. La réponse contient encore tout ce qui a pu être produit. Le champ défaillant est renvoyé avec la valeur null et une entrée dans `errors` indique le problème, sa position dans le document — ligne 8, colonne 5 pour `customer` — et le chemin de la réponse affecté :

```
{
  "data": {
    "order": {
      "status": "open",
      "items": [{ "name": "notebook", "quantity": 2 }],
      "customer": null
    }
  },
  "errors": [
    {
      "message": "Customer service unavailable",
      "locations": [{ "line": 8, "column": 5 }],
      "path": ["order", "customer"]
    }
  ]
}
```

Comme `customer` est déclaré nullable, le dommage reste limité à ce champ. Avec `Customer!`, l'erreur se serait propagée au parent et la commande entière aurait été null. Et puisque `data` n'est pas null, le serveur répond par un statut 2xx malgré l'erreur : HTTP n'a pas de code pour un succès partiel. Un client regardant seulement le statut traitera cette réponse comme complète.

Le modèle inverse, où le serveur détermine la forme de chaque réponse, est celui qu'un projet de ce portfolio a construit. Dans son passage direct de JSON de Postgres à l'API, les fonctions PostgreSQL assemblent la réponse entière en JSON et le gestionnaire Go la transmet sans la modifier. Changer le retour d'une fonction change donc ce que reçoit le client sans toucher une ligne du

gestionnaire. Dans ce projet, le frontend dépend directement de la forme donnée par la base. Dans GraphQL, le client nomme les champs et le serveur doit pouvoir résoudre toute combinaison autorisée par le schéma.

Où les problèmes surviennent

Un champ, mille appels à la base. Les résolveurs sont de petites fonctions, une par champ, qu'un service naïf appelle objet par objet. Une liste de commandes avec leur client coûte alors un appel pour la liste et un autre pour chaque commande. La GraphQL Foundation parle du problème N+1 et préconise le regroupement : les chargements demandés dans une courte fenêtre sont rassemblés et envoyés à la base en une requête, avec un outil comme DataLoader de Facebook.

Une requête demande tout. Le client écrivant l'opération, un document peut imbriquer des amis d'amis sur de nombreux niveaux, répéter un champ coûteux sous cent alias ou réunir beaucoup d'opérations dans une requête. Tout est du GraphQL valide et peut monopoliser le serveur. Les défenses relèvent du service : paginer les champs de liste, refuser avant exécution les opérations dépassant une profondeur ou une largeur maximale, plafonner les alias et la taille des lots. Pour un contrôle plus fin, on pondère les champs et refuse toute opération dont le coût estimé dépasse le budget : c'est l'analyse de complexité des requêtes, pour laquelle la spécification ne donne elle-même aucune règle. Une API réservée à ses propres clients peut aller plus loin avec des documents de confiance : seules les opérations stockées d'avance et retrouvées par hachage peuvent s'exécuter.

200 OK cache l'échec. Une réponse contenant des erreurs peut garder un statut 2xx, comme ci-dessus. Avec l'ancien type de média `application/json`, les recommandations vont plus loin : un statut 2xx est préconisé même lorsqu'une requête valide ne peut pas s'exécuter. Certains anciens serveurs renvoient aussi 2xx pour des erreurs de validation. Le client et sa supervision doivent lire `errors`, pas seulement le statut. Le type plus récent `application/graphql-response+json` redonne du sens au statut : une requête valide ne pouvant pas s'exécuter reçoit alors 4xx ou 5xx. Les clients devraient le demander dans leur en-tête `Accept`.

Le cache ne fonctionne plus. Chaque opération est un POST vers la même URL : les caches entre client et serveur, CDN compris, voient une adresse et une méthode qu'ils ne mettent pas en cache sans indication explicite. Envoyer les

requêtes de lecture par GET aide, mais un long document peut dépasser la longueur d'URL acceptée par les navigateurs et CDN. Les requêtes persistées résolvent les deux problèmes : le client envoie un hachage et le serveur retrouve le document complet avant validation et exécution. Partout où les réponses sont mises en cache sur HTTP, les réponses sensibles doivent rester dans des caches privés ou ne pas être mises en cache.

Un champ disparaît. Les clients choisissent les champs individuellement : en supprimer ou renommer un casse tous ceux qui le sélectionnent encore, et brutalement. Une opération nommant un champ inconnu échoue entièrement à la validation, sans aucune donnée. La spécification garde les champs dépréciés valides précisément pour ne pas casser les clients existants. Ajoutez d'abord le remplaçant, marquez l'ancien avec `@deprecated` et une raison désignant le nouveau, puis supprimez-le seulement quand plus rien ne le sélectionne.

Pour aller plus loin

- Versions de la spécification GraphQL — Toutes les éditions, de juillet 2015 à septembre 2025, plus le brouillon de travail : langage, types, validation, exécution et erreurs.
- Servir sur HTTP — L'URL unique, POST et GET, les deux types de média de réponse et les règles de statut de chacun.
- Performances — GET pour les requêtes de lecture, requêtes persistées, problème N+1 et regroupement.
- Sécurité — Documents de confiance, limites de profondeur, largeur et lots, analyse de complexité, introspection et détails d'erreur en production.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que GraphQL ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/graphql/