Engineer

GraphQL

мова запитів, у якій клієнт сам обирає потрібні поля

Що це таке

GraphQL — це мова запитів для API разом із середовищем виконання, яке обробляє ці запити на сервері. Клієнт надсилає документ, що називає саме ті поля, які йому потрібні, і сервер відповідає даними такої самої форми. Специфікація описує його як мову й рушій виконання для клієнт-серверних програм. У ній зазначено, що GraphQL створено 2012 року, що роботу над ним як над відкритим стандартом почали 2015 року і що з 2019 року він є одним із результатів роботи GraphQL Specification Project при Joint Development Foundation. Найновіше видання вийшло у вересні 2025 року.

Усе починається зі схеми, яку публікує сервіс: набору типів, кожен з іменованими полями, а кожне поле має власний тип. Вбудовані скалярні типи — `Int`, `Float`, `String`, `Boolean` і `ID`, а сервіс може додати власні скаляри, наприклад для UUID. Будь-який тип допускає null, якщо його не позначено `!`, тож `String!` ніколи не буде null, а списки записують у квадратних дужках: `[LineItem!]!` — це список, який завжди присутній і ніколи не містить null. Запити можна надсилати й до самої схеми. Через поля інтроспекції `__schema` і `__type` клієнт може спитати сервіс, що той пропонує, і саме на цьому побудовано інструменти розробника та клієнтські бібліотеки.

Операції бувають трьох видів. Операція `query` — це, словами специфікації, «a read-only fetch», тобто вибірка лише для читання; `mutation` — це запис, за яким іде вибірка; а `subscription` — це довготривалий запит, що доставляє дані у відповідь на послідовність подій у часі. На сервері за кожним полем стоїть функція, яку зазвичай називають резолвером, і вона обчислює значення поля. Сервіс спершу перевіряє операцію на відповідність схемі й лише потім викликає резолвери для полів, які операція вибрала. Поля верхнього рівня в `mutation` виконуються одне за одним, а не паралельно, тож два записи в одному запиті не можуть наввипередки змінювати ті самі дані.

Специфікація не приписує ні транспорту, ні формату серіалізації. Найпоширеніший вибір — HTTP, і тоді GraphQL API доступний за однією URL-адресою, зазвичай із закінченням `/graphql`, і відповідає в JSON, найпоширенішому форматі. Клієнт надсилає POST, у JSON-тілі якого є текст `query` і за потреби `variables`. Сервер може також приймати GET для операцій `query`, але `mutation` завжди надсилають через POST. У відповіді результат лежить у `data`, а все, що пішло не так, — в `errors`. Настанови GraphQL Foundation щодо всього цього спираються на специфікацію GraphQL-over-HTTP, яка досі є чернеткою.

Застосуйте поняття до рішення

Застосуйте «Що таке GraphQL?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Та частина, якої користувач не бачить: де зберігаються дані, які діють правила і як інші системи можуть із нею взаємодіяти.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Спроєктувала й підтримувала 6 пайплайнів ETL/ELT — 6 пайплайнів ETL на трьох рушіях баз даних

Спроєктувала й підтримувала 6 пайплайнів ETL/ELT Зв'язатися з нами →

Чому це важливо

У більшості HTTP API сервер вирішує, що повертає кожна кінцева точка. Екран, якому потрібні статус замовлення та ім’я покупця, або отримує все замовлення й усі дані покупця, тобто більше, ніж просив, або мусить надіслати по запиту на кожен ресурс і сам зшити відповіді. Специфікація називає ці дві проблеми over-fetching і under-fetching. GraphQL робить навпаки: клієнт запитує на рівні окремих полів і отримує саме ці поля, не більше й не менше. Один запит також може йти за зв’язками, від замовлення до його позицій і далі до покупця, бо запит має ту саму форму, що й дані, які він отримає.

Система типів окупається двічі. Кожну операцію перевіряють на відповідність схемі ще до запуску будь-якого резолвера, тож поле з помилкою в назві чи аргумент неправильного типу відхиляються цілком, ще до виконання будь-чого, а інструменти можуть ловити ті самі помилки ще під час написання коду. І оскільки клієнт отримує лише вибрані ним поля, нове поле нічого не коштує наявним клієнтам. Тому API зручно розвивати без номерів версій: нове поле з’являється поруч зі старим, старе позначають `@deprecated` із поясненням причини, воно лишається чинним, щоб наявні клієнти не зламалися, і його прибирають, коли жоден клієнт ним більше не користується.

Ціна реальна, і здебільшого вона виникає тому, що доводиться виходити за межі механізмів самого HTTP. Сутності більше не ідентифікуються через URL, і все йде на одну адресу, зазвичай через POST, тоді як кешованими за замовчуванням зазвичай вважаються саме запити GET. Кешування, яке REST API отримує від посередників задарма, доводиться відбудовувати навмисно. Код стану вже не розповідає всієї історії, бо відповідь, у якій є і дані, і помилки, на рівні HTTP однаково вважається успішною. Тепер клієнт складає запити, яких сервер ніколи не бачив, тож один-єдиний запит може вимагати дуже багато роботи, і захищати від цього сервер — завдання сервісу, а не те, що GraphQL робить за нього. Резолвери на рівні полів легко писати, але, написані наївно, вони стають надто балакучими з базою даних, що стоїть за ними.

Як це виглядає на практиці

Невелика частина схеми для замовлень, кожне з позиціями й покупцем:

```
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!
}
```

Екран, що показує статус, позиції та ім’я покупця, надсилає цю операцію на `/graphql` у тілі POST зі змінними `{"id": "7421"}`:

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

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

Припустімо, що замовлення та його позиції отримано, але сервіс за `customer` не працює. Відповідь однаково містить усе, що вдалося отримати. Поле, яке не вдалося обчислити, повертається як null, а запис в `errors` повідомляє, що пішло не так, де в документі це сталося (рядок 8, стовпець 5 — це `customer`) і якого шляху у відповіді це стосується:

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

Оскільки `customer` оголошено таким, що допускає null, шкода зупиняється на цьому полі. Якби воно було `Customer!`, помилка поширилася б на батьківське поле, і все замовлення повернулося б як null. А оскільки `data` не null, сервер відповідає кодом стану 2xx, хоча помилка є: в HTTP немає коду стану для часткового успіху. Клієнт, який дивиться лише на код стану, сприйме цю відповідь як повну.

Протилежний підхід, коли форму кожної відповіді визначає сервер, реалізував один проєкт із портфоліо цього сайту: наскрізний JSON від Postgres до API. Там функції PostgreSQL збирають усю відповідь у JSON, а обробник на Go пересилає її без змін, тож коли змінюється те, що повертає

функція, змінюється й те, що отримує клієнт, і нікому не треба правити жодного рядка коду обробників. Фронтенд там залежить безпосередньо від форми, яку йому дає база даних. У GraphQL поля називає клієнт, і сервер мусить бути готовим видати будь-яку їхню комбінацію, яку дозволяє схема.

Де виникають проблеми

Одне поле — тисяча звернень до бази даних. Резолвери — це маленькі функції, по одній на поле, і наївний сервіс викликає їх по одному об'єкту за раз. Список замовлень із покупцем кожного замовлення тоді коштує одного звернення до бази даних для списку й ще одного для кожного замовлення в ньому. Настанови GraphQL Foundation називають це проблемою N+1 і вказують на пакетування (batching): завантаження, запитані в межах короткого проміжку часу, збирають разом і надсилають до бази даних одним запитом за допомогою інструмента на кшталт DataLoader від Facebook.

Один запит вимагає всього. Оскільки операцію пише клієнт, один документ може вкладати друзів друзів на багато рівнів углиб, повторити дороге поле під сотнею псевдонімів (aliases) або запакувати багато операцій в один запит. Кожен такий документ — валідний GraphQL, і кожен може надовго зайняти сервер. Захищатися має сервіс: розбивайте поля-списки на сторінки, відхиляйте операції, що перевищують максимальну глибину чи ширину, ще до початку виконання, обмежуйте кількість псевдонімів і операцій в одному запиті, а для тоншого контролю призначайте полям вагу й відхиляйте будь-яку операцію, чия оцінена вартість перевищує бюджет. Цей прийом відомий як аналіз складності запитів (query complexity analysis), і сама специфікація не дає для нього жодних правил. API, що обслуговує лише власних клієнтів, може піти далі з довіреними документами (trusted documents): тоді виконуватися можуть лише операції, збережені заздалегідь і знайдені за хешем.

200 OK приховує збій. Відповідь із помилками однаково може мати код 2xx, як у прикладі вище. Зі старішим медіатипом `application/json` настанови йдуть далі: 2xx рекомендовано навіть тоді, коли валідний запит не вдається виконати, а деякі застарілі сервери повертають 2xx і для помилок валідації. Клієнт і моніторинг навколо нього мусять читати `errors`, а не лише код стану. Новіший медіатип `application/graphql-response+json` повертає коду стану сенс, бо валідний запит, який не вдалося виконати, тоді отримує 4xx або 5xx, і клієнтам варто просити саме його в заголовку `Accept`.

Кешування перестає працювати. Кожна операція — це POST на ту саму URL-адресу, тож кеші між клієнтом і сервером, зокрема CDN, бачать одну адресу й метод, який вони за замовчуванням не кешують. Допомагає надсилати операції `query` через GET, але довгий документ може перевищити довжину URL, яку приймають браузери та CDN. Збережені запити (persisted queries) розв'язують обидві проблеми: клієнт надсилає хеш, а сервер знаходить за ним повний документ, перш ніж валідувати й виконати його. Скрізь, де відповіді кешуються через HTTP, чутливі відповіді треба кешувати приватно або не кешувати взагалі.

Поле зникає. Клієнти обирають поля по одному, тож якщо поле прибрати чи перейменувати, ламається кожен клієнт, який досі його вибирає, і ламається не м'яко: операція, що згадує невідоме поле, цілком не проходить валідацію й не дає жодних даних. Специфікація лишає застарілі поля чинними саме для того, щоб не ламати наявних клієнтів. Спершу додайте заміну, позначте старе поле `@deprecated` із причиною, що вказує на нове, і прибирайте старе поле лише тоді, коли його вже ніщо не вибирає.

Що почитати далі

- GraphQL Specification Versions — усі видання специфікації, від липня 2015 року до видання вересня 2025 року, плюс робоча чернетка: мова, система типів, валідація, виконання й помилки.
- Serving over HTTP — одна кінцева точка, POST і GET, два медіатипи відповідей і правила кодів стану для кожного з них.
- Performance — GET для операцій `query`, persisted queries, проблема N+1 і пакетування.
- Security — trusted documents, обмеження глибини, ширини й кількості операцій у запиті, query complexity analysis, а також інтроспекція й деталі помилок у продакшені.

Словник

Обирайте за фактичними умовами

Опишіть робочий процес, наявні інтерфейси та користувачів, яким потрібна зміна. Ми можемо визначити перевірюваний результат і спосіб передачі, який дасть вашій команді змогу підтримувати його.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке GraphQL?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

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

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/graphql/