Engineer

GraphQL

Abfragesprache, bei der der Client die Felder auswählt

Was es ist

GraphQL ist eine Abfragesprache für APIs samt der Laufzeitumgebung, die diese Abfragen auf dem Server ausführt. Der Client sendet ein Dokument mit genau den gewünschten Feldern. Der Server antwortet mit entsprechend geformten Daten. Die Spezifikation beschreibt GraphQL als Sprache und Ausführungsengine für Client-Server-Anwendungen. Sie hält fest, dass GraphQL 2012 entstand, die Arbeit daran als offenem Standard 2015 begann und es seit 2019 ein Ergebnis des GraphQL Specification Project unter der Joint Development Foundation ist. Die jüngste Veröffentlichung ist die Ausgabe vom September 2025.

Alles beginnt mit dem veröffentlichten Schema eines Dienstes: Typen mit benannten Feldern, die jeweils wiederum einen eigenen Typ haben. Die eingebauten Skalartypen sind `Int`, `Float`, `String`, `Boolean` und `ID`. Ein Dienst kann eigene ergänzen, etwa für eine UUID. Jeder Typ ist nullable, solange er nicht mit `!` markiert ist. Ein `String!` kann daher niemals null sein. Listen stehen in eckigen Klammern: `[LineItem!]!` ist eine stets vorhandene Liste, die niemals null enthält. Auch das Schema lässt sich abfragen. Über die Introspektionsfelder `__schema` und `__type` kann ein Client erfragen, was ein Dienst anbietet. Darauf bauen Entwicklerwerkzeuge und Clientbibliotheken auf.

Es gibt drei Arten von Operationen. Eine Query ist laut Spezifikation „ein rein lesender Abruf“, eine Mutation ein Schreibvorgang mit anschließendem Abruf, eine Subscription eine langlebige Anfrage, die auf eine Folge von Ereignissen im Zeitverlauf Daten liefert. Auf dem Server steht hinter jedem Feld eine Funktion, meist Resolver genannt, die seinen Wert erzeugt. Der Dienst validiert eine Operation zunächst gegen das Schema. Erst dann ruft er die Resolver der ausgewählten Felder auf. Die obersten Felder einer Mutation laufen nacheinander statt parallel. Zwischen zwei Schreibvorgängen derselben Anfrage kann dadurch keine Race Condition entstehen.

Die Spezifikation schreibt weder Transport noch Serialisierungsformat vor. Meist wird HTTP gewählt. Eine GraphQL-API liegt dann an einer einzigen URL, gewöhnlich mit `/graphql` am Ende, und antwortet im verbreitetsten Format JSON. Der Client sendet einen POST, dessen JSON-Body den Abfragetext unter `query` und bei Bedarf die `variables` enthält. Für Queries darf ein Server auch GET akzeptieren; eine Mutation muss jedoch immer POST verwenden. Die Antwort enthält das Ergebnis unter `data`, Fehler unter `errors`. Die Hinweise der GraphQL Foundation hierzu folgen einer GraphQL-over-HTTP-Spezifikation, die weiterhin ein Entwurf ist.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist GraphQL?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Der Teil, den Nutzer nie sehen: wo die Daten liegen, welche Regeln gelten und wie andere Systeme darauf zugreifen dürfen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. 6 ETL/ELT-Pipelines entworfen und betrieben — 6 ETL-Pipelines über drei Datenbankengines hinweg

6 ETL/ELT-Pipelines entworfen und betrieben Kontakt aufnehmen →

Warum es wichtig ist

Bei den meisten HTTP-APIs legt der Server fest, was jeder Endpunkt zurückgibt. Ein Bildschirm, der den Status einer Bestellung und den Kundennamen braucht, erhält entweder die ganze Bestellung und den ganzen Kunden, also mehr als angefragt, oder muss eine Anfrage je Ressource stellen und die Antworten zusammensetzen. Die Spezifikation nennt diese Probleme Over-Fetching und Under-Fetching. GraphQL kehrt die Anordnung um: Der Client fragt auf der Ebene einzelner Felder und erhält genau diese zurück, keine weiteren. Eine Anfrage kann zudem Beziehungen von einer Bestellung zu ihren Positionen und ihrem Kunden verfolgen, weil sie dieselbe Form wie die erwarteten Daten hat.

Das Typsystem zahlt sich zweimal aus. Jede Operation wird gegen das Schema validiert, bevor ein Resolver läuft. Ein falsch geschriebener Feldname oder ein Argument falschen Typs führt deshalb zur vollständigen Ablehnung, bevor irgendetwas ausgeführt wurde. Werkzeuge können dieselben Fehler schon beim Schreiben erkennen. Da ein Client zudem nur die ausgewählten Felder erhält, ist für bestehende Clients wegen eines neuen Felds keine Anpassung nötig. Eine API lässt sich so praktikabel ohne Versionsnummern entwickeln: Ein neues Feld erscheint neben dem alten. Das alte wird mit `@deprecated` und einer Begründung markiert, bleibt gültig, damit bestehende Clients nicht ausfallen, und wird entfernt, sobald es kein Client mehr nutzt.

Die Nachteile sind handfest und entstehen vor allem dadurch, dass GraphQL HTTP-eigene Mechanismen verlässt. Entitäten werden nicht mehr durch URLs identifiziert. Alles geht an eine Adresse, meist per POST, während normalerweise GET-Anfragen standardmäßig als cachebar gelten. Das Zwischenspeichern durch Intermediäre, das eine REST-API ohne Zusatzaufwand erhält, muss bewusst neu aufgebaut werden. Der Statuscode erzählt nicht mehr die ganze Geschichte: Eine Antwort mit Daten und Fehlern zugleich ist auf HTTP-Ebene weiterhin ein Erfolg. Der Client stellt nun Anfragen zusammen, die der Server noch nie gesehen hat. Eine einzige kann daher sehr viel Arbeit verlangen. Den Server davor zu schützen ist Aufgabe des Dienstes, nicht etwas, das GraphQL für ihn übernimmt. Resolver pro Feld sind leicht zu schreiben, führen bei naiver Umsetzung jedoch viele einzelne Aufrufe an die dahinterliegende Datenbank aus.

Wie es in der Praxis aussieht

Ein kleiner Schemaausschnitt für Bestellungen mit Positionen und einem Kunden:

```
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!
}
```

Ein Bildschirm mit Status, Positionen und Kundenname sendet diese Operation im Body eines POST an `/graphql`, mit `{"id": "7421"}` als Variablen:

```
query OrderSummary($id: ID!) {
  order(id: $id) {
    status
    items {
      name
      quantity
    }
    customer {
      name
    }
  }
}
```

Angenommen, Bestellung und Positionen lassen sich auflösen, aber der Dienst hinter `customer` ist ausgefallen. Die Antwort enthält weiterhin alles, was erzeugt werden konnte. Das fehlgeschlagene Feld ist null. Ein Eintrag in `errors` erklärt den Fehler, seine Stelle im Dokument — Zeile 8, Spalte 5 ist `customer` — und den betroffenen Pfad in der Antwort:

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

Weil `customer` als nullable deklariert ist, bleibt der Schaden auf dieses Feld begrenzt. Bei `Customer!` hätte sich der Fehler zum übergeordneten Feld fortgesetzt und die ganze Bestellung wäre null zurückgekommen. Und weil `data` nicht null ist, antwortet der Server trotz Fehler mit einem 2xx-Status: HTTP hat keinen Statuscode für Teilerfolg. Ein Client, der nur den Statuscode betrachtet, hält diese Antwort für vollständig.

Das Gegenmodell, bei dem der Server die Form jeder Antwort bestimmt, setzte ein Projekt im Portfolio dieser Website um. Beim unveränderten Durchreichen von JSON aus Postgres zur API bauen PostgreSQL-Funktionen die ganze Antwort als JSON zusammen. Der Go-Handler leitet sie unverändert weiter. Eine geänderte Funktionsantwort verändert damit die beim Client ankommenden Daten, ohne eine Zeile Handlercode anzufassen. Dort hängt das Frontend direkt von der Datenform der Datenbank ab. In GraphQL benennt der Client die Felder. Der Server muss jede vom Schema erlaubte Kombination auflösen können.

Wo es schiefgeht

Ein Feld, tausend Datenbankaufrufe. Resolver sind kleine Funktionen pro Feld. Ein naiver Dienst ruft sie Objekt für Objekt auf. Eine Bestellliste mit dem Kunden jeder Bestellung kostet dann einen Datenbankaufruf für die Liste und einen weiteren pro Bestellung. Die Hinweise der GraphQL Foundation nennen das N+1-Problem und verweisen auf Batching: Innerhalb eines kurzen Zeitfensters angeforderte Ladevorgänge werden gesammelt und als eine Anfrage an die Datenbank geschickt, etwa mit Facebooks DataLoader.

Eine Anfrage verlangt alles. Weil der Client die Operation schreibt, kann ein Dokument Freunde von Freunden über viele Ebenen verschachteln, ein teures Feld unter hundert Aliasen wiederholen oder viele Operationen in eine Anfrage packen. All das ist gültiges GraphQL und kann Serverressourcen binden. Die Abwehr liegt beim Dienst: Listenfelder paginieren, vor der Ausführung Operationen mit zu großer Tiefe oder Breite ablehnen, Aliasanzahl und Batchgröße begrenzen. Für feinere Kontrolle erhalten Felder Gewichte. Operationen mit geschätzten Kosten über dem Budget werden abgewiesen. Für diese Query Complexity Analysis legt die Spezifikation selbst keine Regeln fest. Eine API nur für eigene Clients kann mit vertrauenswürdigen Dokumenten weitergehen: Ausschließlich vorab gespeicherte und per Hash gefundene Operationen dürfen dann überhaupt laufen.

200 OK verbirgt den Fehler. Wie im Beispiel können Antworten mit Fehlern trotzdem einen 2xx-Status tragen. Beim älteren Medientyp `application/json` gehen die Hinweise weiter: Ein 2xx wird selbst dann empfohlen, wenn eine gültige Anfrage nicht ausgeführt werden kann. Manche älteren Server geben sogar bei Validierungsfehlern 2xx zurück. Der Client und sein Monitoring müssen daher `errors` lesen, nicht nur den Status. Der neuere Medientyp `application/graphql-response+json` gibt dem Status wieder Bedeutung: Eine gültige Anfrage, deren Ausführung scheitert, erhält dann 4xx oder 5xx. Clients sollten diesen Typ im `Accept` -Header anfordern.

Das Caching funktioniert nicht mehr. Jede Operation ist ein POST an dieselbe URL. Caches zwischen Client und Server, einschließlich CDN, sehen daher eine Adresse und eine Methode, die sie standardmäßig nicht zwischenspeichern. Queries per GET helfen, aber ein langes Dokument kann die von Browsern und CDNs akzeptierte URL-Länge überschreiten. Persistierte Queries lösen beide Probleme: Der Client sendet einen Hash, über den der Server vor Validierung und Ausführung das vollständige Dokument nachschlägt. Wo Antworten über HTTP gecacht werden, dürfen sensible Antworten nur in privaten Caches oder gar nicht zwischengespeichert werden.

Ein Feld verschwindet. Clients wählen Felder einzeln. Das Entfernen oder Umbenennen eines Feldes führt daher bei jedem Client, der es weiterhin auswählt, zum vollständigen Fehlschlag der betroffenen Operation: Eine Operation mit einem unbekannten Feld scheitert insgesamt an der Validierung und liefert keinerlei Daten. Die Spezifikation lässt veraltete Felder gerade deshalb gültig, damit bestehende Clients nicht ausfallen. Fügen Sie zuerst den Ersatz hinzu. Markieren Sie das alte Feld mit `@deprecated` und einer Begründung, die auf das neue verweist. Entfernen Sie es erst, wenn es nirgends mehr ausgewählt wird.

Weiterführende Lektüre

- Versionen der GraphQL-Spezifikation — Alle Ausgaben von Juli 2015 bis September 2025 sowie der Arbeitsentwurf: Sprache, Typsystem, Validierung, Ausführung und Fehler.
- Bereitstellung über HTTP — Der einzelne Endpunkt, POST und GET, die beiden Antwortmedientypen und ihre jeweiligen Statuscoderegeln.
- Leistung — GET für Queries, persistierte Queries, das N+1-Problem und Batching.
- Sicherheit — Vertrauenswürdige Dokumente, Grenzen für Tiefe, Breite und Batches, Query Complexity Analysis sowie Introspektion und Fehlerdetails in Produktion.



Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist GraphQL?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/graphql/