Engineer

GraphQL

a query language in which the client picks the fields

What it is

GraphQL is a query language for APIs together with the runtime that carries out those queries on the server. The client sends a document that names exactly the fields it wants, and the server answers with data shaped the same way. The specification describes it as a language and an execution engine for client-server applications. It records that GraphQL was created in 2012, that work on it as an open standard began in 2015 and that, since 2019, it has been a deliverable of the GraphQL Specification Project under the Joint Development Foundation. The latest release is the September 2025 edition.

Everything starts from the schema a service publishes: a set of types, each with named fields, and each field with a type of its own. The built-in scalar types are `Int`, `Float`, `String`, `Boolean` and `ID`, and a service may add custom scalars, such as one for a UUID. Every type is nullable unless it is marked with `!`, so a `String!` can never be null, and lists are written in brackets: `[LineItem!]!` is a list that is always present and never holds a null. The schema can also be queried. Through the introspection fields `__schema` and `__type` a client can ask a service what it offers, and developer tools and client libraries are built on that.

There are three kinds of operation. A query is, in the specification's words, "a read-only fetch"; a mutation is a write followed by a fetch; and a subscription is a long-lived request that delivers data in response to a sequence of events over time. On the server each field is backed by a function, usually called a resolver, that produces its value. The service first validates an operation against the schema and only then calls the resolvers for the fields the operation selected. The top-level fields of a mutation run one after another rather than in parallel, so two writes in the same request cannot race each other.

The specification prescribes neither a transport nor a serialisation format. HTTP is the most common choice, and there a GraphQL API is served at a single URL, usually ending in `/graphql`, and answers in JSON, the most common format. The client sends a POST whose JSON body carries the `query` text and, where needed, its `variables`. A server may also accept GET for queries, but a mutation must always be a POST. The response holds the result under `data` and anything that went wrong under `errors`. The GraphQL Foundation's guidance on all this follows a GraphQL-over-HTTP specification that is still a draft.

Use the concept in a decision

Put "What is GraphQL?" to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

In most HTTP APIs the server decides what each endpoint returns. A screen that needs an order's status and its customer's name either receives the whole order and the whole customer, more than it asked for, or has to make one request per resource and stitch the answers together. The specification calls the two problems over-fetching and under-fetching. GraphQL turns the arrangement around: the client asks at the level of single fields and gets exactly those fields back, no more. One request can also follow relations, from an order to its line items to its customer, because the request is shaped like the data it will receive.

The type system pays off twice. Every operation is validated against the schema before any resolver runs, so a misspelt field or an argument of the wrong type is rejected as a whole, before anything has been executed, and tools can catch the same mistakes while the code is still being written. And since a client only ever gets the fields it selected, adding a field costs existing clients nothing. That makes it practical to evolve an API without version numbers: a new field appears alongside the old one, the old one is marked `@deprecated` with a reason, it stays valid so existing clients do not break, and it is removed once no client uses it any more.

The costs are real, and most come from stepping outside HTTP's own machinery. Entities are no longer identified by URLs, and everything goes to one address, usually by POST, while it is GET requests that are typically treated as cacheable by default. The caching a REST API gets from intermediaries for free has to be rebuilt on purpose. The status code stops telling the whole story, because a response that carries data alongside errors is still a success at the HTTP level. The client now composes requests the server has never seen, so a single request can ask for a great deal of work, and protecting the server from that is the service's job, not something GraphQL does for it. Per-field resolvers are easy to write but, written naively, chatty with the database behind them.

What it looks like in practice

A small part of a schema for orders, each with line items and a customer:

```
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!
}
```

A screen that shows the status, the line items and the customer's name sends this operation to `/graphql` in the body of a POST, with `{"id": "7421"}` as its variables:

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

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

Suppose the order and its items resolve but the service behind `customer` is down. The response still carries everything that could be produced. The failed field comes back null, and an entry in `errors` says what went wrong, where in the document it happened (line 8, column 5 is `customer`) and which path in the response it affects:

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

Because `customer` is declared nullable, the damage stops at that field. Had it been `Customer!`, the error would have propagated to the parent, and the whole order would have come back as null. And because `data` is not null, the server answers with a 2xx status even though there is an error: HTTP has no status code for partial success. A client that looks only at the status code will treat this answer as complete.

The opposite design, in which the server owns the shape of every response, is the one a project in this site's portfolio built. In its JSON passthrough from Postgres to the API, PostgreSQL functions assemble the whole response as JSON and the Go handler forwards it untouched, so changing what a function returns

changes what the client receives without anyone editing a line of handler code. There the frontend depends directly on the shape the database gives it. In GraphQL the client names the fields, and the server must be ready to resolve any combination of them that the schema allows.

Where it goes wrong

One field, a thousand database calls. Resolvers are small functions, one per field, and a naive service calls them one object at a time. A list of orders with each order's customer then costs one database call for the list and one more for every order in it. The GraphQL Foundation's guidance calls this the N+1 problem and points to batching: the loads requested within a short window are collected and sent to the database as a single request, with a tool like Facebook's DataLoader.

One request asks for everything. Because the client writes the operation, one document can nest friends of friends many levels deep, repeat an expensive field under a hundred aliases, or pack many operations into a single request. Each is valid GraphQL, and each can tie up the server. The defences sit with the service: paginate list fields, reject operations that exceed a maximum depth or breadth before execution starts, cap aliases and batch size, and for finer control give fields weights and turn away any operation whose estimated cost goes over budget, a technique known as query complexity analysis that the specification itself gives no rules for. An API that serves only its own clients can go further with trusted documents, so that only operations stored in advance, looked up by hash, may run at all.

200 OK hides the failure. A response with errors can still be a 2xx, as in the example above. With the older `application/json` media type the guidance goes further: a 2xx is recommended even when a valid request fails to execute, and some legacy servers return 2xx for validation errors too. A client, and the monitoring around it, has to read `errors` and not only the status. The newer `application/graphql-response+json` media type makes the status meaningful again, since a valid request that fails to execute then gets a 4xx or 5xx, and clients should ask for it in their `Accept` header.

Caching stops working. Every operation is a POST to the same URL, so the caches between client and server, a CDN included, see one address and a method they do not cache by default. Sending queries as GET helps, but a long document can exceed the URL length that browsers and CDNs accept. Persisted queries

solve both problems: the client sends a hash, and the server looks up the full document before validating and executing it. Wherever responses are cached over HTTP, sensitive ones must be cached privately or not at all.

A field disappears. Clients choose fields one by one, so removing or renaming a field breaks every client that still selects it, and not gently: an operation that names an unknown field fails validation as a whole, with no data at all. The specification keeps deprecated fields valid precisely so that existing clients are not broken. Add the replacement first, mark the old field `@deprecated` with a reason that points to the new one, and remove the old field only once nothing selects it any more.

Further reading

- GraphQL Specification Versions — every edition of the specification, from July 2015 to the September 2025 release, plus the working draft: the language, the type system, validation, execution and errors.
- Serving over HTTP — the single endpoint, POST and GET, the two response media types and the status-code rules for each.
- Performance — GET for queries, persisted queries, the N+1 problem and batching.
- Security — trusted documents, depth, breadth and batch limits, query complexity analysis, and introspection and error detail in production.



Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is GraphQL?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

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

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/graphql/