Engineer

GraphQL

ett frågespråk där klienten själv väljer fälten

Vad det är

GraphQL är ett frågespråk för API:er tillsammans med den körmiljö som utför frågorna på servern. Klienten skickar ett dokument som namnger exakt de fält den vill ha, och servern svarar med data i samma form. Specifikationen beskriver det som ett språk och en exekveringsmotor för klient-server-program. Den anger att GraphQL skapades 2012, att arbetet med det som öppen standard inleddes 2015 och att det sedan 2019 är en leverans från GraphQL Specification Project under Joint Development Foundation. Den senaste utgåvan är från september 2025.

Allt utgår från det schema som en tjänst publicerar: en uppsättning typer, var och en med namngivna fält, och varje fält med en egen typ. De inbyggda skalärtyperna är `Int`, `Float`, `String`, `Boolean` och `ID`, och en tjänst kan lägga till egna skalärer, till exempel en för UUID:er. Varje typ får vara null om den inte är märkt med `!`, så en `String!` kan aldrig vara null, och listor skrivs inom hakparenteser: `[LineItem!]!` är en lista som alltid finns och aldrig innehåller null. Även själva schemat går att ställa frågor till. Genom introspektionsfälten `__schema` och `__type` kan en klient fråga en tjänst vad den erbjuder, och utvecklarverktyg och klientbibliotek bygger just på det.

Det finns tre sorters operationer. En `query` är med specifikationens ord ”a read-only fetch”, alltså en hämtning som bara läser; en `mutation` är en skrivning följd av en hämtning; och en `subscription` är en långlivad förfrågan som levererar data som svar på en följd av händelser över tid. På servern står en funktion bakom varje fält, oftast kallad en resolver, och den tar fram fältets värde. Tjänsten validerar först en operation mot schemat och anropar först därefter resolverfunktionerna för de fält som operationen har valt. Toppnivåfälten i en `mutation` körs ett i taget i stället för parallellt, så att två skrivningar i samma förfrågan inte kan krocka med varandra.

Specifikationen föreskriver varken transport eller serialiseringsformat. HTTP är det vanligaste valet, och där ligger ett GraphQL-API på en enda URL, oftast med ändelsen `/graphql`, och svarar i JSON, det vanligaste formatet. Klienten skickar en POST vars JSON-kropp innehåller `query`-texten och vid behov dess `variables`. En server får också ta emot GET för `query`-operationer, men en `mutation` måste alltid skickas som POST. Svaret har resultatet under `data` och allt som gick fel under `errors`. GraphQL Foundations vägledning om allt detta följer en GraphQL-over-HTTP-specifikation som fortfarande är ett utkast.

Använd begreppet i ett beslut

Använd ”Vad är GraphQL?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Den del användarna aldrig ser: där data lagras, vilka reglerna är och hur andra system får kommunicera med den.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Utformade och drev 6 ETL/ELT-pipelines — 6 ETL-pipelines över tre databasmotorer

Utformade och drev 6 ETL/ELT-pipelines Hör av er →

Varför det spelar roll

I de flesta HTTP-API:er är det servern som bestämmer vad varje slutpunkt returnerar. En skärm som behöver en orders status och kundens namn får antingen hela ordern och hela kunden, alltså mer än den bad om, eller måste skicka en förfrågan per resurs och själv foga ihop svaren. Specifikationen kallar de två problemen over-fetching och under-fetching. GraphQL vänder på det: klienten frågar på nivån av enskilda fält och får exakt de fälten tillbaka, varken mer eller mindre. En förfrågan kan också följa relationer, från en order till dess orderrader och vidare till kunden, eftersom förfrågan har samma form som de data den får tillbaka.

Typsystemet lönar sig två gånger. Varje operation valideras mot schemat innan någon resolver körs, så ett felstavat fält eller ett argument av fel typ avvisas i sin helhet innan något alls har utförts, och verktyg kan fånga samma misstag medan koden fortfarande skrivs. Och eftersom en klient bara får de fält den har valt kostar ett nytt fält ingenting för befintliga klienter. Det gör det praktiskt att vidareutveckla ett API utan versionsnummer: ett nytt fält dyker upp bredvid det gamla, det gamla märks med `@deprecated` och en motivering, det förblir giltigt så att befintliga klienter inte går sönder, och det tas bort när ingen klient använder det längre.

Kostnaderna är verkliga, och de flesta kommer av att man kliver utanför det maskineri som HTTP själv har. Entiteter identifieras inte längre av URL:er, och allt går till en enda adress, oftast med POST, medan det är GET-förfrågningar som i regel räknas som cachebara som standard. Den cachning som ett REST-API får gratis från mellanleden måste byggas upp igen med avsikt. Statuskoden berättar inte längre hela historien, för ett svar med både data och fel är fortfarande en framgång på HTTP-nivå. Klienten sätter nu ihop förfrågningar som servern aldrig har sett, så en enda förfrågan kan begära mycket arbete, och att skydda servern mot det är tjänstens uppgift, inte något GraphQL gör åt den. Resolverfunktioner per fält är lätta att skriva, men skrivna naivt blir de mycket pratsamma mot databasen bakom dem.

Så ser det ut i praktiken

En liten del av ett schema för ordrar, var och en med orderrader och en kund:

```
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!
}
```

En skärm som visar status, orderraderna och kundens namn skickar den här operationen till `/graphql` i kroppen på en POST, med `{"id": "7421"}` som variabler:

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

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

Anta att ordern och dess rader hämtas men att tjänsten bakom `customer` ligger nere. Svaret innehåller ändå allt som gick att ta fram. Fältet som misslyckades kommer tillbaka som null, och en post i `errors` talar om vad som gick fel, var i dokumentet det hände (rad 8, kolumn 5 är `customer`) och vilken sökväg i svaret det gäller:

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

Eftersom `customer` är deklarerat så att det får vara null stannar skadan vid det fältet. Hade det varit `Customer!` hade felet fortplantat sig upp till föräldern, och hela ordern hade kommit tillbaka som null. Och eftersom `data` inte är null svarar servern med en 2xx-status trots att det finns ett fel: HTTP har ingen statuskod för delvis framgång. En klient som bara tittar på statuskoden kommer att behandla svaret som komplett.

Den motsatta designen, där servern äger formen på varje svar, är den som ett projekt i portföljen här på webbplatsen byggde: en JSON-väg rakt från Postgres till API:et. Där sätter PostgreSQL-funktioner ihop hela svaret som JSON, och Go-handlern vidarebefordrar det orört, så när man ändrar vad en funktion returnerar

ändras också det klienten tar emot, utan att någon ändrar en enda rad handlerkod. Frontenden är där direkt beroende av den form som databasen ger den. I GraphQL är det klienten som namnger fälten, och servern måste vara redo att leverera vilken kombination av dem som helst som schemat tillåter.

Var det går fel

Ett fält, tusen databasanrop. Varje resolver är en liten funktion, en per fält, och en naiv tjänst anropar dem ett objekt i taget. En lista med ordrar och varje orders kund kostar då ett databasanrop för listan och ett till för varje order i den. GraphQL Foundations vägledning kallar det N+1-problemet och pekar på batchning: de uppslag som begärs inom ett kort tidsfönster samlas ihop och skickas till databasen som en enda förfrågan, med ett verktyg som Facebooks DataLoader.

En förfrågan begär allt. Eftersom det är klienten som skriver operationen kan ett dokument nästla vänners vänner många nivåer ner, upprepa ett dyrt fält under hundra alias eller packa in många operationer i en enda förfrågan. Var och en är giltig GraphQL, och var och en kan binda upp servern. Försvaret ligger hos tjänsten: paginera listfält, avvisa operationer som överskrider ett största djup eller en största bredd innan exekveringen börjar, sätt tak för alias och batchstorlek, och ge för finare kontroll fälten vikter och avvisa varje operation vars uppskattade kostnad går över budget, en teknik som kallas query complexity analysis och som specifikationen själv inte ger några regler för. Ett API som bara betjänar sina egna klienter kan gå längre med trusted documents, så att bara operationer som har sparats i förväg och slås upp via hash får köras över huvud taget.

200 OK döljer felet. Ett svar med fel kan ändå vara en 2xx, som i exemplet ovan. Med den äldre medietypen `application/json` går vägledningen längre: en 2xx rekommenderas även när en giltig förfrågan inte kan utföras, och vissa äldre servrar returnerar 2xx även vid valideringsfel. En klient, och övervakningen runt den, måste läsa `errors` och inte bara statuskoden. Den nyare medietypen `application/graphql-response+json` ger statuskoden mening igen, eftersom en giltig förfrågan som inte kan utföras då får en 4xx eller 5xx, och klienter bör be om den i sitt `Accept` -huvud.

Cachningen slutar fungera. Varje operation är en POST till samma URL, så cacharna mellan klient och server, ett CDN inräknat, ser en enda adress och en metod som de inte cachar som standard. Det hjälper att skicka `query` -

operationer som GET, men ett långt dokument kan överskrida den URL-längd som webbläsare och CDN:er accepterar. Persisted queries löser båda problemen: klienten skickar en hash, och servern slår upp hela dokumentet innan det valideras och utförs. Överallt där svar cachas över HTTP måste känsliga svar cachas privat eller inte alls.

Ett fält försvinner. Klienter väljer fält ett och ett, så om ett fält tas bort eller byter namn går varje klient som fortfarande väljer det sönder, och inte på något skonsamt sätt: en operation som nämner ett okänt fält underkänns i valideringen i sin helhet och ger inga data alls. Specifikationen håller fält märkta för utfasning giltiga just för att befintliga klienter inte ska gå sönder. Lägg till ersättningen först, märk det gamla fältet med `@deprecated` och en motivering som pekar på det nya, och ta bort det gamla fältet först när ingenting längre väljer det.

Läs mer

- GraphQL Specification Versions — alla utgåvor av specifikationen, från juli 2015 till utgåvan från september 2025, plus arbetsutkastet: språket, typsystemet, validering, exekvering och fel.
- Serving over HTTP — den enda slutpunkten, POST och GET, de två medietyperna för svar och reglerna för statuskoder för var och en av dem.
- Performance — GET för `query`-operationer, persisted queries, N+1-problemet och batchning.
- Security — trusted documents, gränser för djup, bredd och batcher, query complexity analysis samt introspektion och feldetaljer i produktion.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är GraphQL?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/graphql/