Engineer

GraphQL

et forespørgselssprog, hvor klienten selv vælger felterne

Hvad det er

GraphQL er et forespørgselssprog til API'er sammen med den runtime, der udfører forespørgslerne på serveren. Klienten sender et dokument, der nævner præcis de felter, den vil have, og serveren svarer med data i samme form. Specifikationen beskriver det som et forespørgselssprog og en eksekveringsmotor til klient-server-applikationer. Den fortæller, at GraphQL blev skabt i 2012, at arbejdet med det som åben standard begyndte i 2015, og at det siden 2019 har været en leverance fra GraphQL Specification Project under Joint Development Foundation. Den seneste udgave er fra september 2025.

Alt udgår fra det skema, en tjeneste offentliggør: et sæt typer, hver med navngivne felter, og hvert felt med sin egen type. De indbyggede skalartyper er `Int`, `Float`, `String`, `Boolean` og `ID`, og en tjeneste kan tilføje sine egne skalarer, for eksempel én til UUID'er. Enhver type kan være null, medmindre den er markeret med `!`, så en `String!` kan aldrig være null, og lister skrives i firkantede parenteser: `[LineItem!]!` er en liste, der altid er til stede og aldrig indeholder null. Skemaet kan også selv forespørges. Gennem introspektionsfelterne `__schema` og `__type` kan en klient spørge en tjeneste, hvad den tilbyder, og udviklerværktøjer og klientbiblioteker bygger netop på det.

Der findes tre slags operationer. En `query` er med specifikationens ord »a read-only fetch«, altså en hentning, der kun læser; en `mutation` er en skrivning efterfulgt af en hentning; og en `subscription` er en langvarig forespørgsel, der leverer data som svar på en række hændelser over tid. På serveren står en funktion bag hvert felt, som regel kaldet en resolver, og den leverer feltets værdi. Tjenesten validerer først en operation mod skemaet og kalder først derefter resolverne for de felter, operationen har valgt. Topniveaufelterne i en mutation køres ét ad gangen i stedet for parallelt, så to skrivninger i samme forespørgsel ikke kan overhale hinanden.

Specifikationen foreskriver hverken en transport eller et serialiseringsformat. HTTP er det mest almindelige valg, og dér ligger et GraphQL-API på én enkelt URL, som regel med endelsen `/graphql`, og svarer i JSON, det mest almindelige format. Klienten sender en POST, hvis brødtekst er JSON med `query`-teksten og om nødvendigt dens `variables`. En server kan også tage imod GET til queries, men en mutation skal altid sendes som POST. Svaret har resultatet under `data` og alt, hvad der gik galt, under `errors`. GraphQL Foundations vejledning om alt dette følger en GraphQL-over-HTTP-specifikation, der stadig er et udkast.

Brug begrebet i en beslutning

Brug »Hvad er GraphQL?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Den del, brugerne aldrig ser: hvor data ligger, hvilke regler der gælder, og hvordan andre systemer må tale med den.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har designet og drevet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tre databasemotorer

Har designet og drevet 6 ETL/ELT-pipelines Kontakt os →

Hvorfor det betyder noget

I de fleste HTTP-API'er er det serveren, der bestemmer, hvad hvert endpoint returnerer. En skærm, der skal bruge en ordres status og kundens navn, får enten hele ordren og hele kunden, altså mere, end den bad om, eller må sende én forespørgsel pr. ressource og selv sætte svarene sammen. Specifikationen kalder de to problemer over-fetching og under-fetching. GraphQL vender det om: klienten spørger på niveau med de enkelte felter og får præcis de felter tilbage, hverken mere eller mindre. Én forespørgsel kan også følge relationer, fra en ordre til dens ordrelinjer og videre til kunden, fordi forespørgslen har samme form som de data, den får tilbage.

Typesystemet betaler sig to gange. Hver operation valideres mod skemaet, før nogen resolver kører, så et felt med en stavefejl eller et argument af forkert type afvises samlet, før noget overhovedet er udført, og værktøjer kan fange de samme fejl, mens koden stadig bliver skrevet. Og da en klient kun får de felter, den har valgt, koster et nyt felt ingenting for de eksisterende klienter. Det gør det praktisk at videreudvikle et API uden versionsnumre: et nyt felt kommer til ved siden af det gamle, det gamle markeres med `@deprecated` og en begrundelse, det forbliver gyldigt, så eksisterende klienter ikke går i stykker, og det fjernes, når ingen klient bruger det længere.

Omkostningerne er reelle, og de fleste kommer af, at man træder uden for HTTP's eget maskineri. Entiteter identificeres ikke længere af URL'er, og alt går til én adresse, som regel med POST, mens det er GET-forespørgsler, der typisk regnes for cachebare som standard. Den caching, et REST-API får forærende fra mellemleddene, skal bygges op igen med vilje. Statuskoden fortæller ikke længere hele historien, for et svar med både data og fejl er stadig en succes på HTTP-niveau. Klienten sammensætter nu forespørgsler, serveren aldrig har set, så en enkelt forespørgsel kan bede om meget arbejde, og at beskytte serveren mod det er tjenestens opgave, ikke noget, GraphQL gør for den. Resolvere pr. felt er lette at skrive, men skrevet naivt bliver de meget snakkesalige over for databasen bag dem.

Sådan ser det ud i praksis

En lille del af et skema for ordrer, hver med ordrelinjer og en kunde:

```
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 skærm, der viser status, ordrelinjerne og kundens navn, sender denne operation til `/graphql` i brødteksten af en POST med `{"id": "7421"}` som variabler:

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

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

Antag, at ordren og dens linjer bliver hentet, men at tjenesten bag `customer` er nede. Svaret indeholder stadig alt, hvad der kunne leveres. Det fejlede felt kommer tilbage som null, og en post i `errors` fortæller, hvad der gik galt, hvor i dokumentet det skete (linje 8, kolonne 5 er `customer`), og hvilken sti i svaret det berører:

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

Fordi `customer` er erklæret, så det kan være null, standser skaden ved det felt. Havde det været `Customer!`, var fejlen forplantet op til forælderen, og hele ordren var kommet tilbage som null. Og fordi `data` ikke er null, svarer serveren med en 2xx-status, selv om der er en fejl: HTTP har ingen statuskode for delvis succes. En klient, der kun ser på statuskoden, vil behandle svaret som komplet.

Det modsatte design, hvor serveren ejer formen på hvert svar, er det, et projekt i sitets portefølje byggede. I dets JSON-passthrough-arkitektur fra Postgres til API'et samler PostgreSQL-funktioner hele svaret som JSON, og Go-handleren sender det videre uændret, så når man ændrer, hvad en funktion returnerer, ændrer man også, hvad klienten modtager, uden at nogen retter en linje handler-

kode. Dér bygger frontenden direkte på den form, databasen giver den. I GraphQL er det klienten, der nævner felterne, og serveren skal være klar til at levere enhver kombination af dem, som skemaet tillader.

Hvor det går galt

Ét felt, tusind databasekald. Resolvere er små funktioner, én pr. felt, og en naiv tjeneste kalder dem ét objekt ad gangen. En liste over ordrer med hver ordres kunde koster så ét databasekald til listen og ét mere for hver ordre i den. GraphQL Foundations vejledning kalder det N+1-problemet og peger på batching: de opslag, der bliver bedt om inden for et kort tidsrum, samles og sendes til databasen som én forespørgsel, med et værktøj som Facebooks DataLoader.

Én forespørgsel beder om alt. Fordi det er klienten, der skriver operationen, kan ét dokument indlejre venners venner mange niveauer ned, gentage et dyrt felt under hundrede aliasser eller pakke mange operationer ind i én forespørgsel. Hver af dem er gyldig GraphQL, og hver af dem kan lægge beslag på serveren. Forsvaret ligger hos tjenesten: paginér listefelter, afvis operationer, der overskrider en maksimal dybde eller bredde, før udførelsen begynder, sæt loft over aliasser og batchstørrelse, og giv for finere kontrol felterne en vægt og afvis enhver operation, hvis anslåede pris overstiger budgettet, en teknik kendt som query complexity analysis, som specifikationen selv ikke giver nogen regler for. Et API, der kun betjener sine egne klienter, kan gå videre med trusted documents, så kun operationer, der er gemt på forhånd og slås op efter hash, overhovedet må køre.

200 OK skjuler fejlen. Et svar med fejl kan stadig være en 2xx, som i eksemplet ovenfor. Med den ældre medietype `application/json` går vejledningen videre: en 2xx anbefales, selv når en gyldig forespørgsel ikke kan udføres, og nogle ældre servere returnerer også 2xx ved valideringsfejl. En klient, og overvågningen omkring den, må læse `errors` og ikke kun statuskoden. Den nyere medietype `application/graphql-response+json` giver statuskoden mening igen, da en gyldig forespørgsel, hvis udførelse mislykkes, så får en 4xx eller 5xx, og klienter bør bede om den i deres `Accept` -header.

Cachingen holder op med at virke. Hver operation er en POST til den samme URL, så de caches, der sidder mellem klient og server, et CDN medregnet, ser én adresse og en metode, de som standard ikke cacher. Det hjælper at sende queries som GET, men et langt dokument kan overskride den URL-længde, som

browsere og CDN'er accepterer. Persisted queries løser begge problemer: klienten sender en hash, og serveren slår det fulde dokument op, før det valideres og udføres. Overalt, hvor svar caches over HTTP, skal følsomme svar caches privat eller slet ikke.

Et felt forsvinder. Klienter vælger felter ét for ét, så hvis et felt fjernes eller omdøbes, går enhver klient, der stadig vælger det, i stykker, og ikke stille og roligt: en operation, der nævner et ukendt felt, fejler valideringen som helhed og giver slet ingen data. Specifikationen holder forældede felter gyldige netop for, at eksisterende klienter ikke skal gå i stykker. Tilføj erstatningen først, markér det gamle felt med `@deprecated` og en begrundelse, der peger på det nye, og fjern først det gamle felt, når intet længere vælger det.

Læs mere

- GraphQL Specification Versions — alle udgaver af specifikationen, fra juli 2015 til udgaven fra september 2025, plus arbejdsudkastet: sproget, typesystemet, validering, udførelse og fejl.
- Serving over HTTP — det ene endpoint, POST og GET, de to medietyper for svar og reglerne for statuskoder for hver af dem.
- Performance — GET til queries, persisted queries, N+1-problemet og batching.
- Security — trusted documents, grænser for dybde, bredde og batches, query complexity analysis samt introspektion og fejldetaljer i produktion.

Ordliste

Vælg ud fra jeres faktiske vilkår

Beskriv arbejdsgangen, de eksisterende grænseflader og de brugere, der har brug for ændringen. Vi kan definere en testbar leverance og en overdragelse, der gør teamet i stand til at vedligeholde løsningen.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er GraphQL?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

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

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/graphql/