Engineer

GraphQL

et spørrespråk der klienten selv velger feltene

Hva det er

GraphQL er et spørrespråk for API-er sammen med kjøretidsmiljøet som utfører spørringene på serveren. Klienten sender et dokument som navngir nøyaktig de feltene den vil ha, og serveren svarer med data i samme form. Spesifikasjonen beskriver det som et språk og en utførelsesmotor for klient-server-programmer. Den opplyser at GraphQL ble laget i 2012, at arbeidet med det som åpen standard begynte i 2015, og at det siden 2019 har vært en leveranse fra GraphQL Specification Project under Joint Development Foundation. Den nyeste utgaven er fra september 2025.

Alt utgår fra skjemaet en tjeneste publiserer: et sett typer, hver med navngitte felt, og hvert felt med sin egen type. De innebygde skalartypene er `Int`, `Float`, `String`, `Boolean` og `ID`, og en tjeneste kan legge til egne skalarer, for eksempel én for UUID-er. Enhver type kan være null med mindre den er merket med `!`, så en `String!` kan aldri være null, og lister skrives i hakeparenteser: `[LineItem!]!` er en liste som alltid er til stede og aldri inneholder null. Man kan også stille spørsmål til selve skjemaet. Gjennom introspeksjonsfeltene `__schema` og `__type` kan en klient spørre en tjeneste hva den tilbyr, og utviklerverktøy og klientbiblioteker bygger nettopp på det.

Det finnes tre slags operasjoner. En `query` er med spesifikasjonens ord «a read-only fetch», altså en henting som bare leser; en `mutation` er en skriving fulgt av en henting; og en `subscription` er en langvarig forespørsel som leverer data som svar på en rekke hendelser over tid. På serveren står det en funksjon bak hvert felt, som regel kalt en resolver, og den leverer verdien til feltet. Tjenesten validerer først en operasjon mot skjemaet og kaller først deretter resolverne for feltene operasjonen har valgt. Feltene på toppnivå i en `mutation` kjøres ett om gangen i stedet for parallelt, så to skrivinger i samme forespørsel ikke kan gå i beina på hverandre.

Spesifikasjonen foreskriver verken transport eller serialiseringsformat. HTTP er det vanligste valget, og der ligger et GraphQL-API på én enkelt URL, som regel med endelsen `/graphql`, og svarer i JSON, det vanligste formatet. Klienten sender en POST der JSON-kroppen inneholder `query`-teksten og ved behov `variables`. En server kan også ta imot GET for `query`-operasjoner, men en `mutation` må alltid sendes som POST. Svaret har resultatet under `data` og alt som gikk galt, under `errors`. GraphQL Foundations veiledning om alt dette følger en GraphQL-over-HTTP-spesifikasjon som fortsatt er et utkast.

Bruk begrepet i en beslutning

Bruk «Hva er GraphQL?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Delen brukerne aldri ser: hvor dataene ligger, hvilke regler som gjelder, og hvordan andre systemer får lov til å snakke med den.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Designet og driftet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tvers av tre databasemotorer

Designet og driftet 6 ETL/ELT-pipelines Ta kontakt →

Hvorfor det er viktig

I de fleste HTTP-API-er er det serveren som bestemmer hva hvert endepunkt returnerer. En skjerm som trenger statusen til en ordre og navnet på kunden, får enten hele ordren og hele kunden, altså mer enn den ba om, eller må sende én forespørsel per ressurs og sette svarene sammen selv. Spesifikasjonen kaller de to problemene over-fetching og under-fetching. GraphQL snur det hele på hodet: klienten spør på nivå med enkeltfelt og får nøyaktig de feltene tilbake, verken mer eller mindre. Én forespørsel kan også følge relasjoner, fra en ordre til ordrelinjene og videre til kunden, fordi forespørselen har samme form som dataene den får tilbake.

Typesystemet lønner seg to ganger. Hver operasjon valideres mot skjemaet før noen resolver kjører, så et felt med skrivefeil eller et argument av feil type avvises som helhet før noe i det hele tatt er utført, og verktøy kan fange de samme feilene mens koden fortsatt skrives. Og siden en klient bare får feltene den har valgt, koster et nytt felt ingenting for eksisterende klienter. Det gjør det praktisk å videreutvikle et API uten versjonsnumre: et nytt felt kommer til ved siden av det gamle, det gamle merkes med `@deprecated` og en begrunnelse, det forblir gyldig så eksisterende klienter ikke går i stykker, og det fjernes når ingen klient bruker det lenger.

Kostnadene er reelle, og de fleste kommer av at man går utenfor maskineriet HTTP selv har. Entiteter identifiseres ikke lenger av URL-er, og alt går til én adresse, som regel med POST, mens det er GET-forespørsler som vanligvis regnes som cachebare som standard. Den cachingen et REST-API får gratis fra mellomleddene, må bygges opp igjen med vilje. Statuskoden forteller ikke lenger hele historien, for et svar med både data og feil er fortsatt en suksess på HTTP-nivå. Klienten setter nå sammen forespørsler serveren aldri har sett, så én enkelt forespørsel kan be om mye arbeid, og å beskytte serveren mot det er tjenestens jobb, ikke noe GraphQL gjør for den. Resolvere per felt er lette å skrive, men skrevet naivt blir de svært pratsomme mot databasen bak dem.

Slik ser det ut i praksis

En liten del av et skjema 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 skjerm som viser statusen, ordrelinjene og navnet på kunden, sender denne operasjonen til `/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 at ordren og linjene blir hentet, men at tjenesten bak `customer` er nede. Svaret inneholder likevel alt som kunne leveres. Feltet som feilet, kommer tilbake som null, og en oppføring i `errors` forteller hva som gikk galt, hvor i dokumentet det skjedde (linje 8, kolonne 5 er `customer`), og hvilken sti i svaret det gjelder:

```
{
  "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 deklarert slik at det kan være null, stopper skaden ved det feltet. Hadde det vært `Customer!`, ville feilen ha forplantet seg opp til forelderen, og hele ordren ville ha kommet tilbake som null. Og fordi `data` ikke er null, svarer serveren med en 2xx-status selv om det finnes en feil: HTTP har ingen statuskode for delvis suksess. En klient som bare ser på statuskoden, vil behandle svaret som komplett.

Det motsatte designet, der serveren eier formen på hvert svar, er det et prosjekt i porteføljen på dette nettstedet bygget: en JSON-passthrough-arkitektur fra Postgres til API-et. Der setter PostgreSQL-funksjoner sammen hele svaret som JSON, og Go-handleren videresender det urørt, så når man endrer hva en

funksjon returnerer, endres også det klienten mottar, uten at noen redigerer en eneste linje handlerkode. Frontenden avhenger der direkte av formen databasen gir den. I GraphQL er det klienten som navngir feltene, og serveren må være klar til å levere enhver kombinasjon av dem som skjemaet tillater.

Hvor det går galt

Ett felt, tusen databasekall. Resolvere er små funksjoner, én per felt, og en naiv tjeneste kaller dem ett objekt om gangen. En liste over ordrer med kunden til hver ordre koster da ett databasekall for listen og ett til for hver ordre i den. GraphQL Foundations veiledning kaller dette N+1-problemet og viser til batching: oppslagene som blir bedt om innenfor et kort tidsrom, samles og sendes til databasen som én forespørsel, med et verktøy som Facebooks DataLoader.

Én forespørsel ber om alt. Fordi det er klienten som skriver operasjonen, kan ett dokument nøste venners venner mange nivåer ned, gjenta et dyrt felt under hundre aliaser eller pakke mange operasjoner inn i én forespørsel. Hver av dem er gyldig GraphQL, og hver av dem kan legge beslag på serveren. Forsvaret ligger hos tjenesten: paginer listefelt, avvis operasjoner som overskrider en maksimal dybde eller bredde, før utførelsen starter, sett tak på aliaser og batchstørrelse, og gi for finere kontroll feltene vekter og avvis enhver operasjon der den anslåtte kostnaden går over budsjettet, en teknikk kjent som query complexity analysis, som spesifikasjonen selv ikke gir noen regler for. Et API som bare betjener sine egne klienter, kan gå lenger med trusted documents, slik at bare operasjoner som er lagret på forhånd og slås opp etter hash, får kjøre i det hele tatt.

200 OK skjuler feilen. Et svar med feil kan likevel være en 2xx, som i eksempelet ovenfor. Med den eldre medietypen `application/json` går veiledningen lenger: en 2xx anbefales selv når en gyldig forespørsel ikke kan utføres, og noen eldre servere returnerer 2xx også ved valideringsfeil. En klient, og overvåkingen rundt den, må lese `errors` og ikke bare statuskoden. Den nyere medietypen `application/graphql-response+json` gir statuskoden mening igjen, siden en gyldig forespørsel som ikke lar seg utføre, da får en 4xx eller 5xx, og klienter bør be om den i `Accept` -headeren sin.

Cachingen slutter å virke. Hver operasjon er en POST til den samme URL-en, så cachene mellom klient og server, et CDN medregnet, ser én adresse og en metode de som standard ikke cacher. Det hjelper å sende `query` -operasjoner

som GET, men et langt dokument kan overskride URL-lengden som nettlesere og CDN-er godtar. Persisted queries løser begge problemene: klienten sender en hash, og serveren slår opp hele dokumentet før det valideres og utføres. Overalt der svar caches over HTTP, må sensitive svar caches privat eller ikke i det hele tatt.

Et felt forsvinner. Klientene velger felt ett for ett, så hvis et felt fjernes eller får nytt navn, går enhver klient som fortsatt velger det, i stykker, og ikke stille og rolig: en operasjon som nevner et ukjent felt, feiler valideringen som helhet og gir ingen data i det hele tatt. Spesifikasjonen holder felt merket for utfasing gyldige nettopp for at eksisterende klienter ikke skal gå i stykker. Legg til erstatningen først, merk det gamle feltet med `@deprecated` og en begrunnelse som viser til det nye, og fjern det gamle feltet først når ingenting lenger velger det.

Les mer

- GraphQL Specification Versions — alle utgavene av spesifikasjonen, fra juli 2015 til utgaven fra september 2025, pluss arbeidsutkastet: språket, typesystemet, validering, utførelse og feil.
- Serving over HTTP — det ene endepunktet, POST og GET, de to medietypene for svar og reglene for statuskoder for hver av dem.
- Performance — GET for `query` -operasjoner, persisted queries, N+1-problemet og batching.
- Security — trusted documents, grenser for dybde, bredde og batcher, query complexity analysis samt introspeksjon og feildetaljer i produksjon.

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er GraphQL?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/graphql/