The Contract

Understanding GraphQL

Jamie Barton

Contents

Preface

I never intended to write about GraphQL, and I certainly never expected to spend years talking about it.

I started building APIs with GraphQL around the time Graphcool was building an incredible ecosystem. Long before I ever recorded a tutorial, they invited me to GraphQL Europe with no strings attached. It was my first real taste of a community that genuinely cared about making developers more productive.

Meeting Lee Byron, the creator of GraphQL, at GraphQL Europe. I'll never forget the GraphQL bowtie.

That eventually led me to create **graphql.wtf**. I didn't start it to make money and I didn't have a content strategy: if I learnt something

new, I recorded a video explaining one idea in six minutes or less, because developer education should respect people's time. The Guild later sponsored the project for over a year, which paid for recording equipment, editing software, and time away from contracting. They never once dictated what I should create. I ended up talking about their tools anyway, because they were some of the best GraphQL tools available and they solved real problems.

Eventually the project became something else. Companies began asking for sponsored videos, and the more those requests arrived, the more I realised I'd drifted away from why I started. graphql.wtf was never supposed to be advertising. It was supposed to be curiosity. So I stopped.

These days I learn differently. AI can usually answer my questions faster than a video ever could, but I still read, and I still write. Writing forces me to organise my thoughts in a way nothing else does: if I can explain something simply, I probably understand it, and if I can't, I need to learn some more. I have enormous respect for the people who still produce incredible technical videos, like Theo, ThePrimeagen, Aaron Francis, and Josh Cirre. Recording is one of the hardest things I've ever done. Writing feels natural.

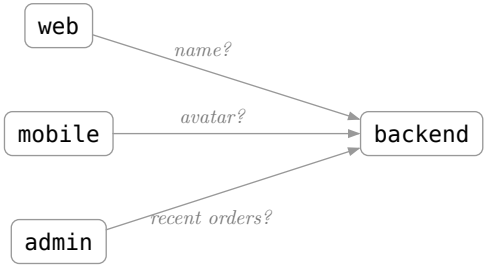
This book isn't a GraphQL tutorial. It's an explanation of the idea that made GraphQL interesting. Not one endpoint. Not fewer requests. Not replacing REST.

A contract.

Chapter 1

The Problem

Imagine three frontend teams: a web application, a mobile app, and an internal admin panel. Each one needs customer information, and each one needs slightly different pieces of it. The web app wants a name, the mobile app wants an avatar, and the admin panel wants recent orders.



One backend serves all three. How do both sides agree on what exists? And how do they evolve together without constantly breaking each other?

That was the problem GraphQL tried to solve.

Chapter 2

A Shared Language

GraphQL isn't a database, an ORM, or a backend framework. It's a language for describing data, and both the client and the server speak it.

The client asks for exactly the shape it wants:

```
{
  customer(id: "1") {
    name
    avatarUrl
  }
}
```

The server answers in the same shape:

```
{
  "customer": {
    "name": "Ada",
    "avatarUrl": "https://example.com/ada.png"
  }
}
```

That shared language becomes the foundation for everything else.

Chapter 3

The Schema

The schema is the most important part of GraphQL. Everything begins here. It describes the data that exists, the relationships between it, and the operations clients can perform:

```
type Customer {
  id: ID!
  name: String!
  avatarUrl: String
  recentOrders: [Order!]!
}

type Query {
  customer(id: ID!): Customer
}
```

Unlike traditional API documentation, the schema can't drift out of date, because it isn't documentation. It *is* the API.

Chapter 4

Types as a Contract

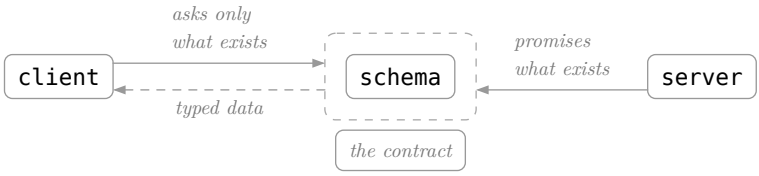
This is where GraphQL became different. Every field has a type, every object has a shape, and every query can be validated before it's ever sent. Ask for something the schema doesn't offer:

```
{
  customer(id: "1") {
    fullName
  }
}
```

and you find out at build time, not in production:

```
Cannot query field "fullName" on type "Customer".
Did you mean "name"?
```

Instead of hoping an API returns what you expect, both sides already know. The backend promises what exists, and the frontend asks only for those things.



That's the contract.

Chapter 5

Clients Without Guesswork

Once you have a contract, tools become possible. GraphQL clients can inspect your schema and generate TypeScript types, API clients, autocomplete, validation, and documentation. The query from Chapter 2 becomes a type without you writing one:

```
type CustomerQuery = {
  customer: {
    name: string
    avatarUrl: string | null
  } | null
}
```

Suddenly your editor understands your API. If the schema changes, your compiler tells you.

Not your users.

Chapter 6

Components That Declare Their Data

This was one of my favourite ideas. Instead of one giant page deciding every piece of data it needed, components could declare their own requirements as fragments:

```
fragment ProfileFields on Customer {
  name
}

fragment AvatarFields on Customer {
  avatarUrl
}

fragment OrdersFields on Customer {
  recentOrders {
    id
    total
  }
}
```

Those individual requirements combine into a single query:

```
query CustomerPage {
  customer(id: "1") {
    ...ProfileFields
    ...AvatarFields
    ...OrdersFields
  }
}
```

Generated types meant every component knew exactly what data it would receive. Pass props, pass types, and stay type-safe all the way down. It felt magical the first time you experienced it.

Chapter 7

The GraphQL Ecosystem

GraphQL wasn't just a specification. It became an ecosystem. Apollo made GraphQL approachable, The Guild built incredible developer tooling, and Prisma made databases feel at home in a GraphQL world. Whether you preferred frontend, backend, or full-stack development, there was a tool for you.

My own journey started with Graphcool, which eventually became Prisma. Without that community, I probably wouldn't have built graphql.wtf.

Chapter 8

REST Was Never the Enemy

One of the biggest misconceptions was that GraphQL wanted to replace REST. It never realistically could: REST is fundamental to the web. GraphQL solved different problems, and for a period of time it was an excellent choice for greenfield applications, especially where multiple clients needed to evolve independently against the same backend.

Then things changed. Backend-for-frontend architectures became popular, frameworks started owning both the frontend and the backend, tRPC appeared, and API routes became first-class citizens. Many of the problems GraphQL solved simply became less common for smaller teams. Meanwhile, enterprise systems kept using GraphQL, because the contract remained valuable.

The technology didn't fail. The industry evolved.

Chapter 9

Observability and Evolution

One thing GraphQL doesn't get enough credit for is observability. Because every query is structured, you know exactly which fields are being requested, and just as importantly, which ones aren't.

That makes evolving an API much easier. Deprecate a field you want rid of:

```
type Customer {
  name: String!
  fullName: String @deprecated(reason: "Use name
instead.")
}
```

Clients migrate, the request count falls to zero, and the field disappears. Instead of guessing what might break, you make decisions based on real usage.

Chapter 10

Where GraphQL Fits Today

GraphQL found its place. It's no longer trying to replace everything, and nor should it. For many applications REST is simpler, for others tRPC is a great fit, and sometimes a framework's built-in API layer is all you need.

But whenever multiple clients, teams, or services need to communicate through a shared contract, GraphQL still shines.

Chapter 11

The Whole Book in One Page

- GraphQL is a language for describing data, spoken by both client and server.
- The schema defines the contract: what exists, how it relates, and what you can ask for. It isn't documentation. It *is* the API.
- The contract enables types, types enable tooling, and tooling enables confidence. Queries are validated before they're ever sent.
- Components declare their own data with fragments and stay type-safe all the way down.
- Structured queries make usage observable: deprecate a field, watch clients migrate, then remove it.
- REST was never the enemy. For one client, use whatever is simplest; for many clients sharing one backend, the contract still shines.

GraphQL didn't become popular because it used one endpoint. It became popular because everyone finally agreed on the same contract.

That's GraphQL.

Now go ask a schema what exists. It'll tell you.