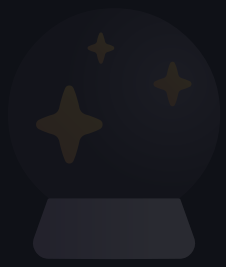

PROGRAMMING

GraphQL

The Type System & Schema · Queries & Resolvers
Mutations & Input Types · The N+1 Problem & DataLoader
Building a Server · Errors & Validation
Authentication & Authorization · Capstone API

Philip Osztromok

Generated with Claude

Table of Contents

GraphQL — 9 Chapters

🔮 Why GraphQL? The REST Problem It Solves

GraphQL isn't "REST, but better" — it's a different answer to a specific set of problems REST APIs run into as an application grows. This chapter is about those problems, using the REST API design already covered in Express (express1-7) and FastAPI as the baseline.

Over-Fetching: More Than the Client Needs

A REST endpoint returns a **fixed shape**. If a UI only needs a user's name, `GET /users/42` still returns everything:

```
// GET /users/42 — the client only wanted "name"
{
 "id": 42,
 "name": "Alice",
 "email": "alice@example.com",
 "address": { /* ...unused... */ },
 "preferences": { /* ...unused... */ },
 "createdAt": "2024-01-15"
}
```

The equivalent GraphQL query asks for exactly what's needed, nothing more:

```
query {
 user(id: 42) {
  name
 }
}
// Response: { "user": { "name": "Alice" } } — nothing else
```

Under-Fetching: Needing More Than One Round Trip

The opposite problem — a UI showing a user's profile plus their recent posts plus each post's comment count needs *multiple* REST calls to assemble one screen:

Three Round Trips vs One Query

REST

`GET /users/42`, then `GET /users/42/posts`, then `GET /posts/{id}/comments/count` for each post — three or more separate requests to build one screen.

GraphQL

One request, nested exactly as deep as the UI needs — the server resolves every level in a single round trip.

One Query, Three Levels Deep

```
query {
 user(id: 42) {
  name
  posts {
   title
   commentCount
  }
 }
}
```

Endpoint Sprawl

Over time, REST APIs tend to accumulate purpose-built endpoints shaped for specific screens: `/users/42/full-profile`, `/users/42/posts/summary`, `/users/42/dashboard-data` — each one a workaround for over/under-fetching on some particular screen. GraphQL replaces all of them with **one endpoint** and lets each client shape its own query.

What GraphQL Actually Is

A query language plus a runtime for schema-based APIs — typically one HTTP endpoint (usually `POST`), where the request body is a query describing exactly the shape of data wanted back.

What It Isn't

Not a database, not automatically faster, and not a wholesale replacement for REST in every case — a complex nested query can still trigger many expensive operations server-side (Chapter 5 covers exactly this).

⚖️ When REST Is Still the Right Choice

This course won't oversell GraphQL as strictly superior. REST remains the better fit for simple CRUD APIs with few clients, for anything leaning on HTTP's built-in `GET`-based caching (harder to get for free with GraphQL's typical single-POST-endpoint shape), for file uploads and simple webhooks, and for teams who don't want the added operational complexity a GraphQL layer introduces.

💻 Coding Challenges

Challenge 1: Identify Over-Fetching

Given a REST endpoint `GET /products/7` that returns 15 fields, and a mobile screen that only displays the product's name and price, explain specifically what's being over-fetched and estimate the wasted payload in general terms.

Goal: Practice recognizing over-fetching in a concrete scenario, not just the abstract definition.

→ Solution

Challenge 2: Write an Under-Fetching Scenario

Describe a realistic UI screen (not from this chapter) that would require at least three separate REST requests to fully populate, and write the single GraphQL query that would replace them.

Goal: Practice translating a multi-endpoint REST scenario into one nested GraphQL query.

→ Solution

Challenge 3: Choose REST or GraphQL

A team is building a simple public webhook receiver with exactly one client and no varying data-shape needs. Recommend REST or GraphQL and justify the choice using this chapter's "when REST still wins" section.

Goal: Practice applying the decision framework honestly, not defaulting to whichever technology is newer.

→ Solution

⚠️ Gotcha: Assuming GraphQL Is Automatically Faster

It's tempting to treat "one request instead of three" as a guarantee of better performance, but GraphQL doesn't make the underlying work cheaper — it just lets the client ask for a deeply nested shape in one request. A single query reaching three levels deep into related data can still trigger just as many (or more) database queries behind the scenes as the REST calls it replaced; Chapter 5 covers exactly this problem. GraphQL also doesn't get HTTP-level `GET` caching for free the way REST does, since queries are typically sent as `POST` requests to one endpoint — caching has to be built deliberately, not assumed.

🎯 What's Next

With the "why" established, the next chapter gets concrete: **The GraphQL Type System & Schema** — the Schema Definition Language, scalar types, and the query/mutation root types every GraphQL API is built around.

📐 The GraphQL Type System & Schema

Chapter 1 established why GraphQL exists. Everything else in this course builds on one artifact: the **schema** — written in the Schema Definition Language (SDL) — which is the explicit, machine-readable contract every GraphQL API is built around.

Scalar Types

GraphQL ships with five built-in scalar types:

Scalar	Meaning
`Int`	A signed 32-bit integer
`Float`	A signed double-precision floating-point value
`String`	UTF-8 text
`Boolean`	`true` or `false`
`ID`	Serialized as a string, but semantically an identifier — not meant to be treated as human-readable text

Object Types

```
type User {
  id: ID!
  name: String!
  email: String
  age: Int
}
```

The `!` means **non-null** — `id` and `name` are guaranteed to always have a value; `email` and `age` may be `null`.

Lists: Where Nullability Gets Confusing

The `!` can apply to the list itself, the items inside it, both, or neither — each combination means something different:

Syntax	Meaning
`[Post]`	The list itself might be `null`; if present, individual items might also be `null`
`[Post!]`	The list itself might be `null`; but if present, every item is guaranteed non-null
`[Post]!`	The list is guaranteed to exist (even if empty); but individual items inside it might be `null`
`[Post!]!`	The list is guaranteed to exist, and every item in it is guaranteed non-null — the strictest, and most common, form

🚪 Query and Mutation: The Entry Points

Two special types define every operation an API supports — nothing is queryable or writable unless it's listed here:

A Small Complete Schema

```
type Query {
  user(id: ID!): User
  posts: [Post!]!
}

type Mutation {
  createPost(title: String!, authorId: ID!): Post!
}

type User {
  id: ID!
  name: String!
  posts: [Post!]!
}

type Post {
  id: ID!
  title: String!
  author: User!
}
```

Query — Reading Only

Every possible top-level read operation is listed as a field on `type Query` — Chapter 1's `user(id: 42) { name }` query is calling the `user` field defined here.

Mutation — Writing Data

Write operations get their own root type. Chapter 4 covers mutations and input types in depth — for now, note that they're just another set of fields, defined the same way as `Query`'s.

Enums

```
enum PostStatus {
  DRAFT
  PUBLISHED
  ARCHIVED
}
```

📝 Schema-First Design

The schema is normally written *before* the resolver functions that implement it — a contract-first approach, similar in spirit to the schema-first validation pattern from the TypeScript Real World Applications course. This lets frontend and backend teams work against an agreed shape in parallel, and lets tools generate documentation, type checking, and even mock servers directly from the schema itself.

Implicit Contract vs Explicit Schema

REST

The "contract" is whatever the JSON response happens to contain — discoverable only through documentation (if it exists and is current) or trial and error.

GraphQL

The schema *is* the contract — machine-readable, and queryable by clients themselves via introspection, which is what powers tools like GraphiQL's autocomplete.

💻 Coding Challenges

Challenge 1: Write a Schema for a Comment System

Write SDL for a `Comment` type (with `id`, `text`, and an `author` field pointing to a `User`) and add a `comments: [Comment!]!` field to the `Post` type from this chapter's example schema.

Goal: Practice writing object type definitions and connecting them to existing types.

→ Solution

Challenge 2: Explain the Four List Nullability Forms

For each of `[Post]`, `[Post!]`, `[Post]!`, and `[Post!]!`, write one sentence describing what a client's code would need to defensively check for that the other three forms wouldn't require.

Goal: Practice reasoning about nullability from the client's perspective, not just the schema's.

→ Solution

Challenge 3: Add an Enum to the Schema

Add a `status: PostStatus!` field to the `Post` type, using the `PostStatus` enum from this chapter, and explain why an enum is a better fit here than a plain `String`.

Goal: Practice choosing the right type for a field with a fixed set of valid values.

→ Solution

⚠️ Gotcha: Getting List Nullability Backwards

It's easy to write `[Post]!` when `[Post!]!` was actually intended, or vice versa — the `!` placement is subtle and the two forms look nearly identical. The practical difference is real: with `[Post]!`, client code technically has to check every individual item in the list for `null` before using it, even though the list itself is guaranteed to exist; with `[Post!]!`, only an empty array is possible, never a `null` entry inside it. Getting this wrong either forces defensive null-checks that should never have been necessary, or — worse — lets a client assume every item is safe to use when the schema never actually guaranteed that.

🎯 What's Next

With the schema's shape defined, the next chapter covers how it actually gets filled with real data: **Queries & Resolvers** — the resolver function signature, and how a query maps down through nested fields.

⚙️ Queries & Resolvers

Chapter 2 defined the schema's *shape*. This chapter is about how a query actually gets *filled* with real data — every field in the schema is backed by a function called a resolver, and understanding how resolvers connect to each other is the key to understanding GraphQL execution.

What a Resolver Is

A resolver is a function attached to one specific field, responsible for returning that field's value. Most scalar fields don't need custom code at all — if a field's name matches a property already present on its parent object, the runtime's **default resolver** just reads it directly.

The Resolver Signature

```
function resolver(parent, args, context, info) { /* … */ }
```

Parameter	What It Is
`parent`	Whatever the *parent field's* resolver returned — not the original query's top-level arguments
`args`	The arguments passed to *this specific field* in the query (e.g. `id: 42`)
`context`	Shared data available to every resolver in one request — the authenticated user, a database connection (Chapter 8 builds on this)
`info`	Metadata about the query itself (field name, path, the full AST) — rarely touched directly

🪆 Resolving Nested Fields, Step by Step

Take this query against Chapter 2's schema:

```
query {
  user(id: 42) {
    name
    posts {
      title
    }
  }
}
```

1. `Query.user`'s resolver runs first, with `args.id === 42`. It looks up and returns a `User` object.
2. For `name`: no custom resolver exists, so the default resolver just reads `parent.name` off the `User` object from step 1.
3. For `posts`: `User.posts`'s resolver runs, with `parent` now being that same `User` object — it typically reads `parent.id` to look up that user's posts.
4. For each returned post's `title`: again, a default resolver reads `parent.title` — `parent` here is now one individual `Post` object from step 3.

Notice that `parent` changes meaning at every level — it's always "whatever the field one level up just returned," never the original top-level `id: 42` argument.

The Resolver Map

```
const resolvers = {
  Query: {
    user: (parent, args, context) => db.users.findById(args.id),
    posts: () => db.posts.findAll(),
  },
  User: {
    posts: (parent) => db.posts.findByAuthorId(parent.id),  // parent = the User
  },
  Post: {
    author: (parent) => db.users.findById(parent.authorId),  // parent = the Post
  },
};
```

When You Need a Custom Resolver

Whenever a field's value isn't already sitting on the parent object with a matching name — a relationship to look up, a computed value, or data pulled from a different source entirely.

When the Default Suffices

Most scalar fields — `User.name`, `Post.title` — need zero resolver code, since the parent object already carries that exact property.

One Handler vs Many Small Functions

REST

One route handler builds and returns the entire response body for an endpoint — all the logic for that shape lives in one function.

GraphQL

Many small, focused resolvers — one per field — compose together automatically into whatever shape the client's query actually asked for.

💻 Coding Challenges

Challenge 1: Write the `Post.comments` Resolver

Using Chapter 2's `Comment` type, write the resolver for `Post.comments` that looks up all comments belonging to a given post, assuming a `db.comments.findByPostId(postId)` function exists.

Goal: Practice writing a resolver that correctly uses `parent` to find related data.

→ Solution

Challenge 2: Trace a Three-Level Query

For the query `{ user(id: 5) { posts { comments { text } } } }`, list each resolver that runs, in order, and state what `parent` is at each step.

Goal: Practice tracing execution through more nesting than the chapter's own worked example.

→ Solution

Challenge 3: Fix a Resolver Using the Wrong Argument

A buggy `User.posts` resolver is written as `(parent, args) => db.posts.findByAuthorId(args.id)`, and it fails whenever `posts` is queried without an `id` argument directly on it. Explain the bug and provide the fix.

Goal: Practice recognizing the exact mistake this chapter's tip box warns about.

→ Solution

⚠️ Gotcha: Confusing `parent` With the Original Query's Arguments

Coming from REST, where one handler has access to the entire request at once, it's natural to expect a nested resolver to somehow still have access to the original top-level arguments (like `id: 42` from the `user` field). It doesn't — a resolver only ever sees its *own* field's `args`, plus whatever `parent` the field one level up returned. A `User.posts` resolver has no `id` argument of its own; it needs to read `parent.id` — the `id` property already present on the `User` object passed down to it — not reach for some argument that was never passed to this specific field at all.

🎯 What's Next

With reading data covered, the next chapter turns to writing it: **Mutations & Input Types** — the `Mutation` root type in practice, input types, and returning updated data after a write.

✏️ Mutations & Input Types

Chapter 3 covered reading data through `Query` resolvers. Writing data uses the exact same resolver mechanics — the `(parent, args, context, info)` signature doesn't change — attached instead to fields on the `Mutation` root type from Chapter 2's schema.

Mutations Are Resolvers Too

The one real spec-level difference: the GraphQL specification guarantees that top-level mutation fields in a single request execute **sequentially**, one after another — unlike query fields, which are free to resolve in parallel. This matters when a mutation's side effects need to happen in a predictable order.

📥 Why Input Types Exist

A regular object `type` (like Chapter 2's `User` or `Post`) can never be used directly as an argument — GraphQL requires a separate, dedicated `input` type for structured arguments:

```
input CreatePostInput {
  title: String!
  authorId: ID!
}

type Mutation {
  createPost(input: CreatePostInput!): Post!
}
```

This isn't just a stylistic convention — a regular `type`'s fields are backed by resolver functions and can reference other types that themselves have resolvers, which makes no sense for something the *client* is sending in as plain data. An `input` type is guaranteed to be pure, resolver-free data, which is exactly what an argument needs to be.

A Complete Mutation, Schema and Resolver

```
// Schema
input CreatePostInput {
  title: String!
  authorId: ID!
}

type Mutation {
  createPost(input: CreatePostInput!): Post!
}

// Resolver
Mutation: {
  createPost: (parent, args, context) => {
    return db.posts.create({
      title: args.input.title,
      authorId: args.input.authorId,
    });
  },
}
```

↩️ Returning Updated Data

Notice `createPost` returns `Post!`, not just a success boolean. This means the client can immediately query fields of the newly created post — including nested ones — in the *same round trip* as the write:

```
mutation {
 createPost(input: { title: "Hello", authorId: 42 }) {
  id
  title
  author { name }
 }
}
```

This is the same "everything in one round trip" idea from Chapter 1's under-fetching problem — just applied to writes instead of reads.

Naming Convention

`verbNoun` — `createPost`, `updatePost`, `deletePost` — makes a schema's write operations immediately scannable.

Payload Wrapper Pattern

Some schemas wrap results in a payload type — `CreatePostPayload { post: Post, errors: [String!] }` — rather than returning the raw type. Chapter 7 builds directly on this for structured error handling.

Separate Endpoints vs Named Operations

REST

`POST /posts`, `PUT /posts/7`, `DELETE /posts/7` — the HTTP method and URL together identify the operation.

GraphQL

One endpoint, distinctly named mutation fields — and the client still chooses exactly which fields come back in the response.

💻 Coding Challenges

Challenge 1: Write an `updatePost` Mutation

Define an `UpdatePostInput` (with `id: ID!` and an optional `title: String`), add an `updatePost` field to `Mutation` returning `Post!`, and write its resolver.

Goal: Practice defining a second input type and mutation following the chapter's `createPost` pattern.

→ Solution

Challenge 2: Write a `deletePost` Mutation

Add a `deletePost(id: ID!): Boolean!` field to `Mutation` and write its resolver, then explain in one sentence why this particular mutation is one of the few reasonable cases for returning a plain `Boolean` instead of the deleted object.

Goal: Practice recognizing when the "return the updated object" convention doesn't apply.

→ Solution

Challenge 3: Explain a Schema Validation Error

A developer writes `createPost(post: Post!): Post!` instead of defining a separate input type, and the schema fails to build. Explain why GraphQL rejects using a regular `type` as an argument.

Goal: Practice explaining the input/output type distinction in your own words.

→ Solution

⚠️ Gotcha: Trying to Reuse a Regular Type as an Argument

It's tempting to reach for an existing `type` (like `Post`) as a mutation argument instead of defining a near-duplicate `input` type — especially when the two look almost identical. GraphQL enforces this separation on purpose: a `type`'s fields are meant to be resolved by functions and can reference other resolver-backed types, which is meaningless for something the client is handing *in* as plain data rather than something the server is computing and handing *out*. There's no way around defining the separate `input` type — the schema simply won't build if a regular `type` is used as an argument.

🎯 What's Next

Writing and reading are both covered — but Chapter 3's step-by-step resolver trace hinted at a real performance trap: **The N+1 Problem & DataLoader**, GraphQL's own version of the N+1 query problem, and how batching fixes it.

🧮 The N+1 Problem & DataLoader

Chapter 3's resolver trace showed that `Post.author`'s resolver runs *once per post* in a list. This chapter is about what that actually costs — GraphQL's own version of the N+1 problem already covered for Django and Rails' ORMs, and how a small utility called DataLoader fixes it.

The Problem, Illustrated

Take this query:

```
query {
  posts {
    title
    author { name }
  }
}
```

1. `Query.posts`'s resolver runs **once** — one query, returning (say) 10 posts.
2. `Post.author`'s resolver then runs **once per post** — 10 separate calls, each issuing its own `db.users.findById(...)` lookup.

Total: **11 queries** (1 + 10) for something that conceptually needs only 2 — one for all the posts, one batched lookup for every author those posts actually reference.

Why This Is Sneakier Than a Typical ORM N+1

A REST/ORM N+1 usually lives inside one endpoint's code a developer can eyeball directly. In GraphQL, the *same* `Post.author` resolver is reused across every query that touches it — its author has no visibility into how many posts any given client's query will actually return, since that's decided entirely by the query shape, not the resolver's own code.

The Fix Isn't "Write Better Resolvers"

A single resolver genuinely can't know in advance how many times it will run in a given request — the fix has to happen at a layer that CAN see all the calls made during one request: batching.

📦 DataLoader: Batching and Caching

DataLoader — built by the same team that created GraphQL — collects every individual `.load(id)` call made during a single tick of the event loop and merges them into **one** batched request:

Setting Up and Using a Loader

```
import DataLoader from "dataloader";

function createUserLoader() {
  return new DataLoader(async (ids) => {
    const users = await db.users.findByIds(ids);  // ONE query for every id collected
return ids.map(id => users.find(u => u.id === id));  // must match input order exactly
  });
}

// Resolver, using the loader from context (Chapter 3's fourth parameter)
Post: {
  author: (parent, args, context) => context.userLoader.load(parent.authorId),
}
```

Naive Resolver vs DataLoader-Batched Resolver

> **Naive**
>
> `db.users.findById(parent.authorId)` runs immediately, once per post — 10 posts, 10 separate queries.

> **DataLoader**
>
> `context.userLoader.load(parent.authorId)` queues the ID; all 10 queued IDs are collected and resolved in exactly one batched call.

Caching Within a Request

DataLoader also caches: calling `.load(42)` twice in the same request returns the cached result the second time, without hitting the batch function again — deduplicating repeated lookups of the

same ID within one query execution.

Batching ≠ Caching

Batching merges *different* IDs requested in the same tick into one call. Caching avoids re-fetching the *same* ID twice. DataLoader does both, but they're genuinely separate mechanisms.

One Loader Per Request, Not Global

A DataLoader instance must be created fresh inside whatever function builds the per-request `context` object — never shared across requests. The tip box below covers exactly why.

💻 Coding Challenges

Challenge 1: Batch the `Post.comments` Resolver

Rewrite Chapter 3's `Post.comments` resolver to use a DataLoader that batches by `postId`, given a `db.comments.findByPostIds(postIds)` function that returns comments for multiple posts in one query.

Goal: Practice writing a batch function whose return array matches the input ID order, when each key maps to a LIST of results rather than a single one.

→ Solution

Challenge 2: Count the Queries, Before and After

For a query returning 25 posts where every post shares one of only 4 distinct authors, state exactly how many database queries the naive resolver runs versus the DataLoader-batched version.

Goal: Practice quantifying the concrete savings, not just describing them abstractly.

→ Solution

Challenge 3: Diagnose a Cross-Request Cache Leak

A codebase creates one `DataLoader` instance as a module-level singleton, shared across every request, instead of creating a new one per request. Explain what can go wrong, especially in an app with per-user data access rules.

Goal: Practice reasoning about why request-scoping matters, not just batching mechanics.

→ Solution

⚠️ Gotcha: A Shared, Global DataLoader Instance

It's tempting to create one `DataLoader` at startup and reuse it for every request — it seems more efficient than constructing a new one each time. It isn't safe: DataLoader's cache has no concept of "which request" or "which user" a cached result belongs to. If User A's request caches a lookup, and User B's request later calls `.load()` with the same ID, User B gets User A's cached result back — potentially leaking data across users, or serving stale data indefinitely since the cache never expires on its own. A DataLoader instance must be created fresh inside whatever function builds the per-request `context` object from Chapter 3, so its cache lives and dies with exactly one request.

🎯 What's Next

With performance handled, the next chapter puts everything together into a real, running server: **Building a GraphQL Server with Node.js** — Apollo Server or graphql-yoga, wiring resolvers and DataLoaders to an actual data source.

 Building a GraphQL Server with Node.js

Every chapter so far has been schema, resolvers, and DataLoader as concepts and snippets. This chapter wires all of it into an actual running server, using the same Node.js and Express foundation already covered in this curriculum.

Choosing a Server Library

Two common options: **Apollo Server** (batteries-included, the most widely adopted) and **graphql-yoga** (lighter, built closely on the spec's reference implementation). This chapter walks through Apollo Server, but every concept here — `typeDefs`, resolvers, context — transfers directly to graphql-yoga or any other spec-compliant server.

Basic Setup

```
npm install @apollo/server graphql
```

```
import { ApolloServer } from "@apollo/server";
import { startStandaloneServer } from "@apollo/server/standalone";

const typeDefs = `#graphql
  type Query {
    user(id: ID!): User
  }
  type User {
    id: ID!
    name: String!
  }
`;

const resolvers = {
  Query: {
    user: (parent, args, context) => context.userLoader.load(args.id),
  },
};

const server = new ApolloServer({ typeDefs, resolvers });
const { url } = await startStandaloneServer(server, {
  async context({ req }) {
    return { userLoader: createUserLoader() };  // Chapter 5 — fresh per request
  },
});
```

🧳 The Context Function, in Practice

This is exactly where Chapter 5's per-request DataLoader instances get created, and where authentication info will go in Chapter 8 — the context function runs once per incoming request and returns whatever object every resolver's third parameter will see.

Wiring Resolvers to a Real Data Source

The `db.users.findById(...)`-style pseudocode from earlier chapters becomes a real database client — the same `mysql2` patterns from Node.js Course 2:

```
function createUserLoader(pool) {
  return new DataLoader(async (ids) => {
    const [rows] = await pool.query(
      "SELECT * FROM users WHERE id IN (?)", [ids]
    );
    return ids.map(id => rows.find(r => r.id === id));
  });
}
```

🔗 Attaching to Express

In a real deployment, GraphQL usually sits inside an existing Express app rather than standing alone — mounted at one path:

A Complete Minimal Server

```
import express from "express";
import { expressMiddleware } from "@apollo/server/express4";

const app = express();
const server = new ApolloServer({ typeDefs, resolvers });
await server.start();

app.use(
  "/graphql",  // one endpoint — Chapter 1's theme, made concrete
  express.json(),
  expressMiddleware(server, {
    async context({ req }) => ({ userLoader: createUserLoader(pool) }),
  })
);

app.listen(4000);
```

Built-In Playground

Apollo Server ships a browser-based GraphiQL-style explorer during development — writing and running queries by hand against the live schema, with autocomplete powered by introspection from Chapter 2.

Splitting `typeDefs` as Schemas Grow

A schema doesn't have to live in one string — larger APIs typically split `typeDefs` across multiple files by domain (users, posts, comments), merged together at startup.

Where the Routing Actually Lives

REST

Routing is scattered across many `app.get`/`post`/`put`/`delete` calls, often spread across multiple route files.

GraphQL

One `app.use("/graphql", ...)` mount point — the schema and resolvers themselves are the real "routing," not Express routes.

💻 Coding Challenges

Challenge 1: Add a Second Loader to Context

Extend the Express-mounted server's `context` function to also create and return a `postsByAuthorLoader` (per Chapter 5's batching pattern), alongside the existing `userLoader`.

Goal: Practice building out a context function that carries multiple DataLoaders at once.

→ Solution

Challenge 2: Mount at a Custom Path With a Health Check

Modify the Express server example so GraphQL is mounted at `/api/graphql` instead of `/graphql`, and add a separate plain REST endpoint `GET /health` returning `{ status: "ok" }`.

Goal: Practice combining a GraphQL endpoint with an ordinary REST route in the same Express app.

→ Solution

Challenge 3: Find the Bug in a Context Function

A developer writes `const userLoader = createUserLoader(pool);` at the TOP of the server file (outside any function), then references that same `userLoader` variable inside the `context` callback instead of calling `createUserLoader(pool)` there. Explain the bug this reintroduces.

Goal: Practice spotting Chapter 5's gotcha in realistic server-setup code, not just in isolated loader snippets.

→ Solution

⚠️ Gotcha: Hoisting Loader Creation Out of the Context Function

This is Chapter 5's gotcha resurfacing in a very concrete place: it's tempting, while wiring up a real server, to create a DataLoader once near the top of the file — outside the `context` callback — to "avoid recreating it on every request." That's precisely the shared-instance mistake Chapter 5 warned about, just easier to accidentally write here because the `context` function is only a few lines and the loader creation looks like it could reasonably live outside it. Every DataLoader must be constructed *inside* the `context` callback itself, so a genuinely fresh instance — and fresh cache — exists for every single request.

🎯 What's Next

With a real server running, the next chapter covers what happens when things go wrong: **Error Handling & Validation** — GraphQL's error format compared to HTTP status codes, and custom error types.

🚨 Error Handling & Validation

Chapter 6 built a working server. This chapter covers what happens when something goes wrong — and GraphQL's approach is genuinely different from the HTTP status codes REST relies on.

Errors Don't Change the HTTP Status

A GraphQL response typically returns **HTTP 200** even when a resolver throws — errors are reported in a separate `errors` array in the response body, alongside a `data` field that might still be partially populated:

```
{
 "data": {
  "user": {
   "name": "Alice",
   "posts": null
  }
 },
 "errors": [
  {
   "message": "Failed to load posts",
   "path": ["user", "posts"],
   "locations": [{ "line": 3, "column": 5 }]
  }
 ]
}
```

Because a query touches many resolvers, some fields can succeed while others fail in the very same response. If `Post`'s `title` field were non-null (`String!`, per Chapter 2) and its resolver threw, that null would propagate *up* to the nearest nullable parent instead — GraphQL's execution guarantees a non-null field is never actually returned as `null`.

The Built-In Error Shape

Field	What It Is
`message`	Human-readable description of what went wrong
`locations`	Line/column in the original query where the failing field was requested
`path`	The exact field path that failed — matches Chapter 3's resolver-nesting structure

🏷️ Custom Errors: Conveying a Category

GraphQL has no equivalent to HTTP's 404/401/500 — instead, an `extensions.code` convention plays that role:

Throwing a Custom Error

```
import { GraphQLError } from "graphql";

Query: {
 user: async (parent, args, context) => {
  const user = await context.userLoader.load(args.id);
  if (!user) {
   throw new GraphQLError("User not found", {
    extensions: { code: "NOT_FOUND" },
   });
  }
  return user;
 },
}
```

Conveying Error Category

> **REST**
>
> The HTTP status code itself carries the category — `404`, `401`, `400`, `500`.

> **GraphQL**
>
> Status stays `200`; category lives in `extensions.code` instead — `NOT_FOUND`, `UNAUTHENTICATED`, `BAD_USER_INPUT`.

Input Validation Is a Separate Concern

The schema's `!` only guarantees a field is *present* and the *right type* — not that a string isn't empty, or a number is in a sensible range. That's the same "never trust input" lesson from the site's security courses, applied to mutation arguments before a write happens:

```
createPost: (parent, args, context) => {
  if (args.input.title.trim().length === 0) {
    throw new GraphQLError("Title cannot be empty", {
      extensions: { code: "BAD_USER_INPUT" },
    });
  }
  return db.posts.create(args.input);
}
```

📦 The Payload Pattern, Revisited

Chapter 4 mentioned wrapping mutation results in a payload type. That pattern is often preferred specifically for *expected, recoverable* validation errors, keeping the top-level `errors` array reserved for unexpected, systemic failures:

Top-Level `errors` Array

Unexpected, systemic problems — a database connection failure, a bug. The client generally can't do anything meaningful except show a generic failure state.

Payload-Level `errors` Field

Expected, user-facing validation problems — "title too short," "email already taken." The client can show these directly next to the relevant form field.

💻 Coding Challenges

Challenge 1: Throw an `UNAUTHENTICATED` Error

Write a resolver for a hypothetical `deletePost` mutation that throws a `GraphQLError` with `extensions.code: "UNAUTHENTICATED"` if `context.currentUser` is missing, before performing the delete.

Goal: Practice throwing a categorized error before a resolver's main logic runs.

→ Solution

Challenge 2: Design a Payload Type With Errors

Define a `CreatePostPayload` type with `post: Post` and `errors: [UserError!]!` fields (where `UserError { field: String!, message: String! }`), and change `createPost`'s return type to it.

Goal: Practice the payload-wrapper pattern for surfacing expected validation errors alongside a possibly-null result.

→ Solution

Challenge 3: Explain a Client Bug

A client's code does `if (response.status === 200) { showSuccess(); }` after every GraphQL mutation, without ever inspecting the response body's `errors` array. Explain what's wrong with this and what the client should do instead.

Goal: Practice recognizing the exact misconception this chapter's tip box addresses.

→ Solution

⚠️ Gotcha: Checking HTTP Status Instead of the Response Body

Client code trained on REST habitually checks `response.ok` or `status === 200` as a proxy for "the operation succeeded" — for GraphQL, that check is meaningless. A `200` response with a populated `errors` array is a completely normal, expected shape, not an edge case. Every GraphQL client must inspect the response body's `errors` array directly, every time, regardless of HTTP status — treating a 200 status alone as proof of success will silently miss real, reported failures sitting right there in the same response.

🎯 What's Next

With errors handled, the next chapter covers who's allowed to do what: **Authentication & Authorization in GraphQL** — context-based auth, and field-level authorization.

🔐 Authentication & Authorization in GraphQL

Chapter 7 covered throwing an `UNAUTHENTICATED` error. This chapter is about implementing auth properly — authentication (who's asking) and authorization (what they're allowed to see) in an API with exactly one endpoint, building on the session/JWT material from the site's Authentication & Session Security course.

Authentication Happens Once, in Context

Unlike REST, where each route can independently apply its own auth middleware, GraphQL has one endpoint — so authentication happens exactly once, inside the context function from Chapter 6:

```
async context({ req }) {
  const authHeader = req.headers.authorization ?? "";
  const token = authHeader.replace("Bearer ", "");
  const currentUser = verifyToken(token);  // null if missing/invalid
return {
    currentUser,
    userLoader: createUserLoader(pool),  // Chapter 5
  };
}
```

Every resolver in the request now sees the same `context.currentUser` — either a real user or `null` — without needing its own separate auth check to establish *who's asking*.

🎯 Authorization Happens Per Field

A single query can mix public and protected fields, so authorization *can't* live in one central gate the way REST middleware does — it belongs inside whichever specific resolvers actually need it:

A Field Only Visible to the Owner or an Admin

```
User: {
  email: (parent, args, context) => {
    const isOwner = context.currentUser?.id === parent.id;
    const isAdmin = context.currentUser?.role === "ADMIN";
    return (isOwner || isAdmin) ? parent.email : null;  // silent null, not an error
  },
}
```

Return `null` vs Throw

Returning `null` for a nullable field lets the rest of the query still succeed — good for "you're just not allowed to see this one thing." Throwing (Chapter 7) is clearer but forces the client to specifically handle the error.

Reusable Authorization Helpers

Rather than repeating checks in every resolver, wrap them: `requireAuth(context)`, `requireOwnerOrAdmin(context, ownerId)` — called at the top of any resolver that needs them.

```
function requireAuth(context) {
  if (!context.currentUser) {
    throw new GraphQLError("Not authenticated", { extensions: { code: "UNAUTHENTICATED" } });
  }
  return context.currentUser;
}

Mutation: {
  deletePost: (parent, args, context) => {
    const user = requireAuth(context);  // reused, not repeated
return db.posts.delete(args.id, user.id);
  },
}
```

One Gate vs Many Small Checks

> **REST**
>
> Auth middleware runs once, before a route handler — the whole endpoint is protected or it isn't.

> **GraphQL**
>
> Checks are scattered across whichever resolvers need them, since one query commonly mixes public and protected fields in a single request.

Some libraries offer schema-level `@auth`-style custom directives as an alternative to hand-written per-resolver checks — worth knowing exists, though this course sticks to explicit resolver-level checks for clarity.

💻 Coding Challenges

Challenge 1: Write `requireOwnerOrAdmin`

Write a reusable `requireOwnerOrAdmin(context, resourceOwnerId)` function that throws a `FORBIDDEN`-coded `GraphQLError` unless `context.currentUser` is either the resource's owner or has an `ADMIN` role.

Goal: Practice generalizing the chapter's inline ownership check into a reusable helper.

→ Solution

Challenge 2: Choose Null vs Throw

For each of (a) a `User.email` field hidden from non-owners, and (b) an `updatePost` mutation called by someone who doesn't own the post, recommend returning `null` or throwing, and justify each choice.

Goal: Practice applying the null-vs-throw tradeoff to two different kinds of operations.

→ Solution

Challenge 3: Find the Unprotected Path

A schema checks authorization only inside `Query.user`'s resolver, assuming this protects every `User` field from unauthorized access. Explain a different query path that reaches the same `User` data without ever running that check.

Goal: Practice recognizing that the same type can be reached through multiple different resolver paths.

→ Solution

⚠️ Gotcha: Protecting One Entry Point Isn't Protecting the Data

It's tempting to add an authorization check to `Query.user`'s resolver and consider `User` data protected — but that's only *one path* to reach a `User` object. `Query.posts` → `Post.author` returns full `User` objects too, through a completely different resolver that never runs `Query.user`'s check at all. In GraphQL, the same type is frequently reachable through many different nested paths, and a check placed on one entry point does nothing to protect any of the others. Authorization needs to live on the *field actually returning sensitive data* (like `User.email` in this chapter's own example) — not just on whichever root query happens to be the "obvious" way in.

🎯 What's Next

Every piece is now in place — the final chapter is a capstone: **Building a Small GraphQL API**, combining the schema, resolvers, mutations, DataLoader, and auth from every chapter into one working service.

🏁 Capstone: Building a Small GraphQL API

Every previous chapter built one piece in isolation. This capstone assembles them into one small blog-style API — `User`, `Post`, `Comment` — with authentication, batched resolvers, and structured errors. Nothing new is introduced here, only assembly.

What's Being Built

- Read posts and their authors/comments in one nested query
- Create and delete posts, restricted to authenticated owners
- Batched, N+1-safe resolvers for every relationship
- Categorized errors the client can act on, not just generic failures

📐 The Schema

```
type User {
  id: ID!
  name: String!
  email: String // Ch.8 — resolver hides this unless owner/admin
  posts: [Post!]!
}

type Post {
  id: ID!
  title: String!
  author: User!
  comments: [Comment!]!
}

type Comment {
  id: ID!
  text: String!
  author: User!
}

input CreatePostInput { title: String! }
type UserError { field: String!, message: String! }
type CreatePostPayload { post: Post, errors: [UserError!]! }

type Query {
  posts: [Post!]!
  post(id: ID!): Post
}

type Mutation {
  createPost(input: CreatePostInput!): CreatePostPayload!
  deletePost(id: ID!): Boolean!
}
```

🖥️ Server: Everything Combined

```
async context({ req }) {  // Ch.6 + Ch.8
const token = (req.headers.authorization ?? "").replace("Bearer ", "");
 return {
  currentUser: verifyToken(token),
  userLoader: createUserLoader(pool),        // Ch.5
  commentsByPostLoader: createCommentsByPostLoader(pool),  // Ch.5
 };
}


const resolvers = {
 Query: {
  posts: () => db.posts.findAll(),
  post: (parent, args) => db.posts.findById(args.id),
 },
 Mutation: {
  createPost: (parent, args, context) => {  // Ch.4 + Ch.7 + Ch.8
const user = requireAuth(context);
    if (args.input.title.trim().length === 0) {
     return { post: null, errors: [{ field: "title", message: "Title cannot be empty" }] };
    }
    const post = db.posts.create({ title: args.input.title, authorId: user.id });
    return { post, errors: [] };
  },
  deletePost: (parent, args, context) => {
   const post = db.posts.findById(args.id);
   requireOwnerOrAdmin(context, post.authorId);  // Ch.8
return db.posts.delete(args.id);
  },
 },
 User: {
  email: (parent, args, context) =>              // Ch.8 — field-level auth
   (context.currentUser?.id === parent.id) ? parent.email : null,
  posts: (parent) => db.posts.findByAuthorId(parent.id),
 },
 Post: {
  author: (parent, args, context) => context.userLoader.load(parent.authorId),  // Ch.5
comments: (parent, args, context) => context.commentsByPostLoader.load(parent.id), // Ch.5
 },
};
```

Every Field Auth-Protected Individually

`User.email` is checked at the FIELD, not at whatever query happened to reach it — exactly the fix for Chapter 8's "unprotected path" gotcha, since this same resolver runs whether the User came from a direct query or via `Post.author`.

What's Simplified Here

No real database, no pagination, no rate limiting — this capstone stays honest about scope, focusing on how the pieces from Chapters 2–8 fit together, not on production hardening.

Was GraphQL Worth It for This API?

Honestly — Yes, Here

Posts/authors/comments is a genuinely nested, multi-shaped read pattern — exactly Chapter 1's under-fetching problem.

But Not Automatically

A flatter API with one client and no nested reads (Chapter 1's webhook example) would gain nothing from this — the decision was never "GraphQL is better," only "it fits this shape."

💻 Coding Challenges

Challenge 1: Add a `createComment` Mutation

Add a `CreateCommentInput { postId: ID!, text: String! }`, a `createComment` mutation returning `Comment!`, and a resolver requiring authentication before creating the comment.

Goal: Practice extending the capstone's schema and resolvers with a new, consistent operation.

→ Solution

Challenge 2: Verify the Auth Fix Actually Works

Trace the query `{ posts { author { email } } }` against this chapter's resolvers, for a caller who isn't the author, and confirm `email` comes back `null` rather than leaking the value — citing exactly which resolver prevents it.

Goal: Practice verifying Chapter 8's gotcha is genuinely closed, not just described.

→ Solution

Challenge 3: Trace a Full Mutation Lifecycle

Narrate, step by step, everything that happens from an authenticated client calling `createPost(input: { title: "" })` to the client receiving its response — naming which chapter's mechanism handles each step.

Goal: Practice holding the entire course's material together as one coherent system.

→ Solution

⚠️ Gotcha: Testing Has to Cover Query Shape, Not Just Endpoints

A REST API's tests map cleanly onto its endpoints — test `GET /posts`, test `POST /posts`, done. A GraphQL API has exactly one endpoint but effectively unlimited query *shapes*, and Chapter 8's capstone made this concrete: the SAME `User.email` resolver behaves differently depending on whether the caller is the owner, an admin, or neither — and it's reachable through multiple different query paths (`Query.user`, `Post.author`, `Comment.author`). Testing this API properly means testing representative query *shapes* and permission combinations, not just "does the endpoint respond" — a single resolver can be correct in isolation and still leak data through a query shape nobody thought to test.

🎓 Course Complete

This closes out the **GraphQL** course — from the specific REST problems it solves, through the schema and type system, resolvers and mutations, the N+1 problem and DataLoader, a real Node.js server, structured error handling, authentication and authorization, and finally this capstone tying every piece into one working API.