November 14, 2017

GraphQL Server Basics: GraphQL Schemas, TypeDefs & Resolvers Explained

Nikolas Burk
@nikolasburk

Structure and implementation of GraphQL servers (Part I)

When starting out with GraphQL — one of the first questions to ask is *how do I build a GraphQL server*? As GraphQL has been released simply as a *specification*, your GraphQL server can literally be *implemented* in any of your preferred programming languages.

Before starting to build your server, GraphQL requires you to design a *schema* which in turn defines the API of your server. In this post, we want to understand the schema's major components, shed light on the mechanics of actually implementing it and learn how libraries, such as GraphQL.js, `graphql-tools` and `graphene-js` help you in the process.

> *This article only touches on **plain GraphQL** functionality — there's no notion of a network layer defining **how** the server communicates with a client. The focus is on the inner workings of a "GraphQL execution engine" and the query resolution process. To learn about the network layer, check out the next article.*

The GraphQL schema defines the server's API

Defining schemas: The Schema Definition Language

GraphQL has its own type language that's used the write GraphQL schemas: The Schema Definition Language (SDL). In its simplest form, GraphQL SDL can be used to define types looking like this:

```graphql
type User {
  id: ID!
  name: String
}
```

The `User` type alone doesn't expose any functionality to client applications, it simply defines the structure of a user *model* in your application. In order to add functionality to the API, you need to add fields to the root types of the GraphQL schema: `Query`, `Mutation` and `Subscription`. These types define the *entry points* for a GraphQL API.

For example, consider the following query:

```graphql
query {
  user(id: "abc") {
    id
    name
  }
}
```

This query is only valid if the corresponding GraphQL schema defines the `Query` root type with the following `user` field:

```graphql
type Query {
  user(id: ID!): User
}
```

So, the schema's root types determine the shape of the queries and mutations that will be accepted by the server.

> *The GraphQL schema provides a clear contract for client-server communication.*

The `GraphQLSchema` object is the core of a GraphQL server

GraphQL.js is Facebook's reference implementation of GraphQL and provides the foundation for other libraries, like `graphql-tools` and `graphene-js`. When using any of these libraries, your development process is centered around a `GraphQLSchema` object, consisting of two major components:

- the schema *definition*
- the actual *implementation* in the form of resolver functions

For the example above, the `GraphQLSchema` object looks as follows:

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: {
    id: { type: GraphQLID },
    name: { type: GraphQLString },
  },
})

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      user: {
        type: UserType,
        args: {
          id: { type: GraphQLID },
        },
      },
    },
  }),
})
```

As you see, the SDL-version of the schema can be directly translated into a JavaScript representation of type `GraphQLSchema`. Note that this schema doesn't have any resolvers — it thus wouldn't allow you to actually *execute* any queries or mutations. More about that in the next section.

Resolvers implement the API

Structure vs Behaviour in a GraphQL server

GraphQL has a clear separation of *structure* and *behaviour*. The *structure* of a GraphQL server is — as we just discussed — its schema, an abstract description of the server's capabilities. This structure comes to life with a concrete *implementation* that determines the server's *behaviour*. Key components for the implementation are so-called *resolver* functions.

> ***Each field in a GraphQL schema is backed by a resolver.***

In its most basic form, a GraphQL server will have *one* resolver function *per field* in its schema. Each resolver knows how to fetch the data for its field. Since a GraphQL query at its essence is just a collection of fields, all a GraphQL server actually needs to do in order to gather the requested data is invoke all the resolver functions for the fields specified in the query. (This is also why GraphQL often is compared to RPC-style systems, as it essentially is a language for invoking remote functions.)

Anatomy of a resolver function

When using GraphQL.js, each of the fields on a type in the `GraphQLSchema` object can have a `resolve` function attached to it. Let's consider our example from above, in particular the `user` field on the `Query` type — here we can add a simple `resolve` function as follows:

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      user: {
        type: UserType,
        args: {
          id: { type: GraphQLID },
        },
        resolve: (root, args, context, info) => {
          const { id } = args // the `id` argument for this field is declared above
          return fetchUserById(id) // hit the database
        },
        },
      },
    }),
  })
```

Assuming a function `fetchUserById` is actually available and returns a `User` instance (a JS object with `id` and `name` fields), the `resolve` function now enables *execution* of the schema.

Before we dive deeper, let's take a second to understand the four arguments passed into the resolver:

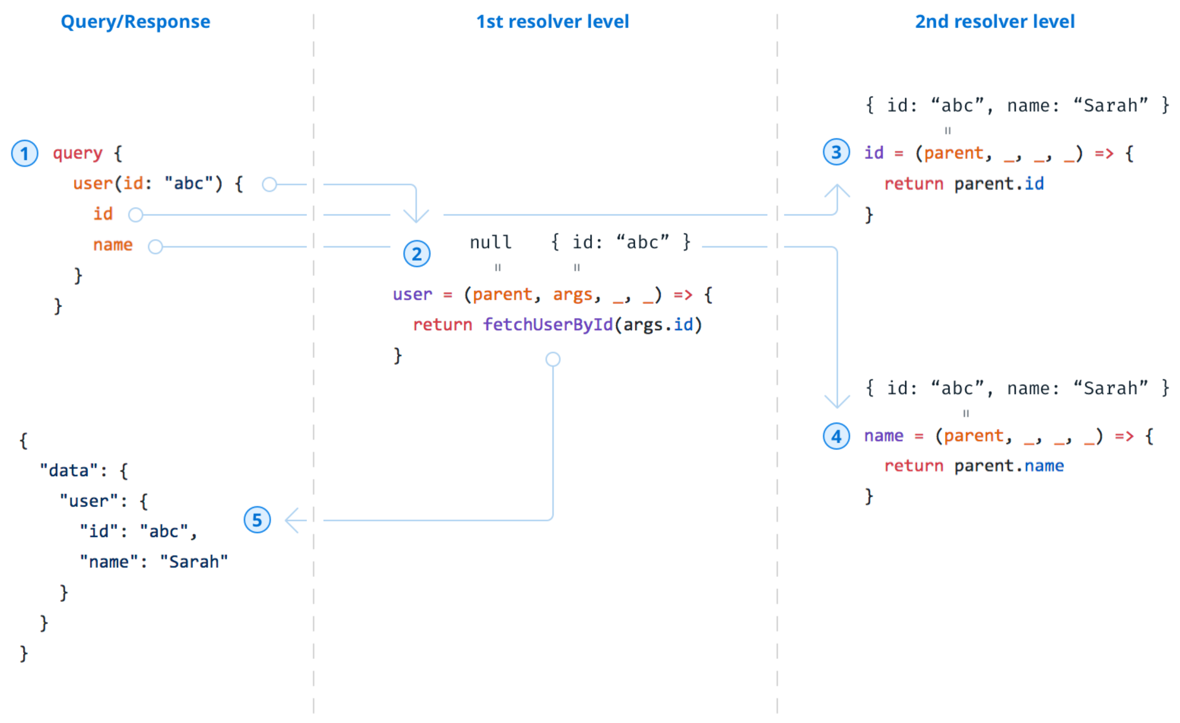
1. `root` (also sometimes called `parent`): Remember how we said all a GraphQL server needs to do to resolve a query is calling the resolvers of the query's fields? Well, it's doing so *breadth-first* (level-by-level) and the `root` argument in each resolver call is simply the result of the previous call (initial value is `null` if not otherwise specified).
2. `args`: This argument carries the parameters for the query, in this case the `id` of the `User` to be fetched.
3. `context`: An object that gets passed through the resolver chain that each resolver can write to and read from (basically a means for resolvers to communicate and share information).
4. `info`: An AST representation of the query or mutation. You can read more about the details in part III of this series: Demystifying the info Argument in GraphQL Resolvers.

Earlier we stated that *each field in the GraphQL schema is backed by a resolver function*. For now we only have one resolver, while our schema in total has three fields: the root field user on the `Query` type, plus the `id` and `name` fields on the `User` type. The two remaining fields still need their resolvers. As you'll see, the implementation of these resolvers is trivial:

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: {
    id: {
      type: GraphQLID,
      resolve: (root, args, context, info) => {
        return root.id
      },
    },
    name: {
      type: GraphQLString,
      resolve: (root, args, context, info) => {
        return root.name
      },
    },
  },
})
```

Query execution

Considering our query from above, let's understand how it's executed and data is collected. The query in total contains three fields: `user` (the *root field*), `id` and `name`. This means that when the query arrives at the server, the server needs to call three resolver functions — one per field. Let's walk through the execution flow:



```
query {
  user(id: "abc") {
    id
    name
  }
}
```

```
user = (parent, args, _, _) => {
  return fetchUserById(args.id)
}
```

```
id = (parent, _, _, _) => {
  return parent.id
}
```

```
name = (parent, _, _, _) => {
  return parent.name
}
```

```
{
  "data": {
    "user": {
      "id": "abc",
      "name": "Sarah"
    }
  }
}
```

1. The query arrives at the server.

2. The server invokes the resolver for the root field `user` — let's assume `fetchUserById` returns this object: `{ "id": "abc", "name": "Sarah" }`
3. The server invokes the resolver for the field `id` on the `User` type. The `root` input argument for this resolver is the return value from the previous invocation, so it can simply return `root.id`.
4. Analogous to 3, but returns `root.name` in the end. (Note that 3 and 4 can happen in parallel.)
5. The resolution process is terminated — finally the result gets wrapped with a `data` field to adhere to the GraphQL spec:

```
{
  "data": {
    "user": {
      "id": "abc",
      "name": "Sarah"
    }
  }
}
```

Now, do you really need to write resolvers for `user.id` and `user.name` yourself? When using GraphQL.js, you don't have to implement the resolver if the implementation is as trivial as in the example. You can thus omit their implementation since GraphQL.js already infers what it needs to return based on the names of the fields and the root argument.

Optimizing requests: The DataLoader pattern

With the execution approach described above, it's very easy to run into performance problems when clients send deeply nested queries. Assume our API also had *articles* with *comments* to ask for and allowed for this query:

```
query {
  user(id: "abc") {
    name
    article(title: "GraphQL is great") {
      comments {
        text
        writtenBy {
          name
        }
      }
    }
  }
}
```

Notice how we're asking for a specific `article` from a given `user`, as well as for its `comments` and the `name`s of the users who wrote them.

Let's assume this article has five comments, all written by the same user. This would mean we'd hit the `writtenBy` resolver five times but it would just return the same data every time. The DataLoader allows you to optimize in these kinds of situations to avoid the N+1 query problem — the general idea is that resolver calls are batched and thus the database (or other data source) only has to be hit once.

> *To learn more about the DataLoader, you can watch this excellent video by Lee Byron: DataLoader — Source code walkthrough (~35 min)*

GraphQL.js vs `graphql-tools`

Now let's talk about the available libraries that help you implement a GraphQL server in JavaScript — mostly this is about the difference between GraphQL.js and `graphql-tools`.

GraphQL.js provides the foundation for graphql-tools

The first key thing to understand is that GraphQL.js provides the foundation for `graphql-tools`. It does all the heavy lifting by defining required types, implementing schema building as well as query validation and resolution. `graphql-tools` then provides a thin convenience layer on top of GraphQL.js.

Let's take a quick tour through the functions that GraphQL.js provides. Note that its functionality is generally centered around a `GraphQLSchema`:

- `parse` and `buildASTSchema`: Given a GraphQL schema defined as a *string* in GraphQL SDL, these two functions will create a GraphQLSchema instance: `const schema = buildASTSchema(parse(sdlString))`.
- `validate`: Given a `GraphQLSchema` instance and a query, `validate` ensures the query adheres to the API defined by the schema.
- `execute`: Given a `GraphQLSchema` instance and a query, `execute` invokes the resolvers of the query's fields and creates a response according to the GraphQL specification. Naturally, this only works if resolvers are part of a `GraphQLSchema` instance (otherwise it's just restaurant with a menu but without a kitchen).
- `printSchema`: Takes a `GraphQLSchema` instance and returns its definition in the SDL (as a *string*).

Note that the most important function in GraphQL.js is `graphql` which takes a `GraphQLSchema` instance and a query — and then calls `validate` and `execute`:

```
graphql(schema, query).then(result => console.log(result))
```

> *To get a sense of all these functions, take a look at this simple node script that uses them in a straightforward example.*

The `graphql` function is executing a GraphQL query against a schema which in itself already contains *structure* as well as *behaviour*. The main role of `graphql` thus is to orchestrate the invocations of the resolver functions and package the response data according to the shape of the provided query. In that regard, the functionality implemented by the `graphql` function is also referred to as a *GraphQL engine*.

`graphql-tools`: Bridging interface and implementation

One of the benefits when using GraphQL is that you can employ a *schema-first* development process, meaning every feature you build first manifests itself in the GraphQL schema — then gets implemented through corresponding resolvers. This approach has many benefits, for example it allows frontend developers to start working against a mocked API, before it is actually implemented by backend developers— thanks to the SDL.

> ***The biggest shortcoming of***

GraphQL.js is that it doesn’t allow you to write a schema in

the SDL and then easily generate an executable version of a

`GraphQLSchema`.

As mentioned above, you can create a `GraphQLSchema` instance from SDL using `parse` and `buildASTSchema`, but this lacks the required `resolve` functions that make execution possible! The only way for you to make your `GraphQLSchema` executable (with GraphQL.js) is by manually adding the `resolve` functions to the schema's fields.

`graphql-tools` fills this gap with one important piece of functionality: `addResolveFunctionsToSchema`. This is very useful as it can be used to provide a nicer, SDL-based API for creating your schema. And that's precisely what `graphql-tools` does with `makeExecutableSchema`:

```
const { makeExecutableSchema } = require('graphql-tools')

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

const resolvers = {
  Query: {
    user: (root, args, context, info) => {
      return fetchUserById(args.id)
    },
  },
}

const schema = makeExecutableSchema({
  typeDefs,
  resolvers,
})
```

So, the biggest benefit of using `graphql-tools` is its nice API for connecting your declarative schema with resolvers!

When not to use `graphql-tools`?

We just learned that `graphql-tools` at its core provides a convenience layer on top of GraphQL.js, so are there cases when it's not the right choice for implementing your server?

As with most abstractions, `graphql-tools` makes certain workflows easier by sacrificing flexibility somewhere else. It offers an amazing "Getting Started"-experience and avoids friction when quickly building up a `GraphQLSchema`. If your backend has more custom requirements though, such as dynamically constructing and modifying your schema, its corset might be a bit too tight — in which case you can just fall back to using GraphQL.js.

A quick note on `graphene-js`

`graphene-js` is a new GraphQL library following the ideas from its Python counterpart. It also uses GraphQL.js under the hood, but doesn't allow for schema declarations in the SDL.

`graphene-js` deeply embraces modern JavaScript syntax, providing an intuitive API where queries and mutations can be implemented as JavaScript classes. It's very exciting to see more GraphQL implementations coming up to enrich the ecosystem with fresh ideas!

Conclusion

In this article, we unveiled the mechanics and inner workings of a GraphQL execution engine. Starting with the GraphQL schema which defines the API of the server and determines what queries and mutations will be accepted, and what the response format has to look like. We then went deep into resolver functions and outlined the execution model enabled by a GraphQL engine when resolving incoming queries. Finally ending up with an overview of the available JavaScript libraries that help you implement GraphQL servers.

> *If you want to get a practical overview of what was discussed in this article, check out this repository. Notice that it has a `graphql-js` and `graphql-tools` branch to compare the different approaches.*

Generally, it's important to note that GraphQL.js provides all the functionality you need for building GraphQL servers — `graphql-tools` simply implements a convenience layer on top that caters most use cases and provides a great "Getting Started"-experience. Only with more advanced requirements for building your GraphQL schema, it might make sense to take the gloves off and use plain GraphQL.js.

In the next article, we'll discuss the network layer and different libraries for implementing GraphQL servers like express-graphql, apollo-server and graphql-yoga. Part 3 then covers the structure and role of the info object in GraphQL resolvers.

Join the discussion

Follow @prisma on Twitter

Twitter

Don't miss the next post!

Sign up for the Prisma newsletter

your@email.com

Join

Fullstack App With TypeScript, PostgreSQL, Next.js, Prisma & GraphQL: Data Modeling

August 18, 2021

This article is the first part of a course where we build a fullstack app with Next.js, GraphQL, TypeScript,Prisma and PostgreSQL. In this article, we'll create the data model and explore the different components of Prisma.

Prisma 2.27 Adds Preview Support for MongoDB

July 14, 2021

Today we are excited to expand the range of supported databases with preview support of **MongoDB**. Try it out and let us know what you think!

Prisma

PRODUCTS

Prisma Client

Prisma Migrate

Prisma Studio

Prisma 1 Cloud

Prisma Data Platform

Product Roadmap

RESOURCES

Docs

Get Started

API Reference

Examples

How to GraphQL

Data Guide

Enterprise Event

PRISMA WITH

Prisma with Next.js

Prisma with GraphQL

Prisma with Apollo

Prisma with NestJS

Prisma with Express

Prisma with hapi

COMMUNITY

Prisma Ambassador

Meet the Community

Prisma Day

Slack

GitHub

Discussions

GraphQL Meetup

TypeScript Meetup

Advanced TypeScript Trickery

Connect Dev Africa