Intro to Server GraphQL with Node.js

Scott Moss
Frontend Masters

GraphQL, the big picture

What is GraphQL?

A spec that describes a declarative query language that your clients can use to ask an API for the exact data they want. This is achieved by creating a strongly typed Schema for your API, ultimate flexibility in how your API can resolve data, and client queries validated against your Schema.

It’s just a spec. There are several implementations and variations

Server Side

- Type Definitions
- Resolvers
- Query Definitions
- Mutation Definitions
- Composition
- Schema

Client Side

- Queries
- Mutations
- Fragments

Where does GraphQL fit in?

- A GraphQL server with a connected DB (most greenfields)
- A GraphQL server as a layer in front of many 3rd party services and connects them all with one GraphQL API
- A hybrid approach where a GraphQL server has a connected DB and also communicates with 3rd party services

Node GraphQL Tools

- Servers
  - Apollo server
  - GraphQL Yoga
  - … Others
- Services
  - Amplify
  - … others
- Tools
  - Prisma
  - … many many others

Schemas

Creating a Schema

- Using Schema Definition Language (SDL)
- Programmatically Creating a Schema using language constructs

Use the SDL to create your Schemas

Basic parts

- **Types** - a construct defining a shape with fields
- **Fields** - keys on a Type that have a name and a value type
- **Scalars** - primitive value type built into GraphQL
- **Query** - type that defines how clients can access data
- **Mutation**- type that defines how clients can modify or create data

Hello World GraphQL API

Follow along

Query Type

What is a Query?

A **Type** on a Schema that defines operations clients can perform to access data that resembles the shape of the other Types in the Schema.

Creating Queries

- Create Query Type in the Schema using SDL
- Add fields to the Query Type
- Create Resolvers that for the fields

What are Resolvers?

Functions that are responsible for returning values for fields that exist on Types in a Schema. Resolvers execution is dependent on the incoming client Query.

Creating Resolvers

- Resolver names must match the exact field name on your Schema’s Types
- Resolvers must return the value type declared for the matching field
- Resolvers can be async
- Can retrieve data from any source

Schema + Resolvers ⇒ Server

To create a server, at minimum, we need a Query Type with a field, and a Resolver for that field.

Your turn

- Define a Query Type in your Schema
- Define fields on your Query Type
- Create Resolvers for the fields on your Query Type
- Create a server from your Schema and Resolvers

Arguments and Input Types

Arguments

- Allows clients to pass variables along with Queries that can be used in your Resolvers to get data
- Must be defined in your Schema
- Can be added to any field
- Either have to be Scalars or Input Types

Input Type

- Just like Types, but used for Arguments
- All field value types must be other Input Types or Scalars

Arguments in Resolvers

- Arguments will be passed to field Resolvers as the second argument
- The argument object will strictly follow the argument names and field types
- Do whatever you want with them

Your turn

- Create input Types for Query arguments
- Add arguments for your Queries
- Use arguments in your Query field resolvers to filter data

Mutation Type

What are Mutations?

A **Type** on a Schema that defines operations clients can perform to mutate data (create, update, delete).

Creating Mutations

- Define Mutation Type on Schema using SDL
- Add fields for Mutation type
- Add arguments for Mutation fields
- Create Resolvers for Mutation fields

Return values for Mutation fields

- Dependent on your clients and use case
- If using a client side GraphQL cache, you should return the exact values Queries return

Your turn

- Define a Mutation Type in your Schema
- Add fields on your Mutation Type
- Create Input Types for your Mutation field arguments
- Create Resolvers for your Mutation Fields

Advanced SDL

Enums

A set of discrete values that can be used in place of Scalars. An enum field must resolve to one of the values in the Enum. Great for limiting a field to only a few different options.

Interfaces

Abstract Types that can’t be used as field values but instead used as foundations for explicit Types. Great for when you have Types that share common fields, but differ slightly.

Unions

Like interfaces, but without any defined common fields amongst the Types. Useful when you need to access more than one disjoint Type from one Query, like a search.

Relationships

Thinking in Graphs

Your API is no longer a predefined list of operations that always return the same shapes. Instead, your API is a set of Nodes that know how to resolve themselves and have links to other Nodes. This allows a client to ask for Nodes and then follow those links to get related Nodes.

Adding Relationships

- Add a Type as a field value on another Type
- Create resolvers for those fields on the Type

Your turn

- Add new fields on Types the reference other Types
- Create resolvers for field Types that point to Types

GraphQL, the big picture

What is GraphQL?

A spec that describes a declarative query language that your clients can use to ask an API for the exact data they want. This is achieved by creating a strongly typed Schema for your API, ultimate flexibility in how your API can resolve data, and client queries validated against your Schema.

It’s just a spec. There are several implementations and variations

Server Side

- Type Definitions
- Resolvers
- Query Definitions
- Mutation Definitions
- Composition
- Schema

Client Side

- Queries
- Mutations
- Fragments

Where does GraphQL fit in?

- A GraphQL server with a connected DB (most greenfields)
- A GraphQL server as a layer in front of many 3rd party services and connects them all with one GraphQL API
- A hybrid approach where a GraphQL server has a connected DB and also communicates with 3rd party services

Queries and Mutations from the client

Operation names

Unique names for your client side Query and Mutation operations. Used for client side caching, indexing inside of tools like GraphQL playground, etc. Like naming your functions in JS vs keeping them anonymous.

Variables with operations

Operations can define arguments, very much like a function in most programming languages. Those variables can then be passed to query / mutation calls inside the operation as arguments. Variables are expected to be given at run time during operation execution from your client.

Apollo Client

What is Apollo Client

Encapsulates HTTP logic used to interact with a GraphQL API. Doubles as a client side state management alternative as well. If your GraphQL API is also an Apollo Server, provides some extra features. Offers a plug approach for extending its capabilities. It’s also framework independent.

Storing data from your API

- All nodes are stored flat by an unique ID
- Unique ID is defaulted to .id or ._id from nodes. You can change this
- Every node should send an .id or ._id, or none at all. Or you have to customize that logic

Queries in React

Mutations in React

Keeping Cache in Sync

Why is the cache out of sync?

If you perform a mutation that updates or creates a single node, then apollo will update your cache automatically given the mutation and query has the same fields and id.

If you perform a mutation that updates a node in a list or removes a node, you are responsible for updating any queries referencing that list or node. There are many ways to do this with apollo.

Keeping cache in sync

- Refetch matching queries after a mutation
- Use update method on mutation
- Watch Queries

Optimistic UI

What is a Optimistic UI?

Your UI does not wait until after a mutation operation to update itself. Instead, it anticipates the response from the API and proceeds as if the API call was sync. The the API response replaces the generated one. This gives the illusion of your being really fast.

Optimistic UI with mutations

Apollo provides a simple hook that allows you to write to the local cache after a mutation.

Client Side Schemas

Why?

In addition to managing data from your API, apollo client can also local state originated from your front end app. Stuff you would normally store in something like Redux or Vuex. You can create a schema to define that state which allows you to query for that state the same way you query your API for data.

How?

The exact same way as the server. You just have to extend the Types from your server schema. You then use a directive to access local state from your queries and mutations.

You made it 💯