GraphQL:Thinking in Resolvers

Sébastien Lavoie-Courchesne

February 23rd 2022

Don't know how your clients are using your APIs?

Just return everything. . . the right way!

Sébastien Lavoie-Courchesne

Architect for the Catalog Group at AppDirect

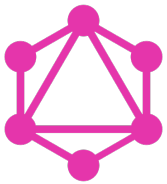
Member of the GraphQL Working Group

Developed most of the initial infrastructure we use

Contents

Technologies used in the examples

GraphQL



NodeJS

TypeScript

ApolloServer

Chirper

A chirping service

Create a user

Send chirps

Reply to chirps

GraphQL

What is GraphQL?

From the GraphQL SpecificationGraphQL-contributors, *GraphQL Specification, October 2021 Edition*:

> *GraphQL is a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions.*

But What is GraphQL?

Specification

Query language

Communication over the network

Server and client requirements

Why GraphQL?

Only request and receive the fields you want:

- Smaller payloads
- Descriptive language
- Documented

How does it work?

Server defines the schema it serves using the GraphQL language

Clients can read the schema and send queries to the server using the GraphQL language, optionally passing in variables

Operations

Query
: a read-only fetch, analogous to REST GET

Mutation
: a write followed by a fetch, analogous to REST POST/PATCH/PUT/DELETE

Subscription
: a long-lived request that fetches data in response to source events

HTTP Communication

Most common use of GraphQL
HTTP GET or POST to a specific URL
Traditional HTTP headers/cookies
Returns status code 200

GET `query` parameter contains the query as a string
POST JSON Body includes:
`query`, `variables`, `operationName`

Server

Servers handle requests by:

- parsing query string
- building execution plan
- executing plan if valid
- returning result

Example Request

```
{
  "query": "
    query ($id: ID!) {
      chirp(id: $id) {
        contents
      }
    }
  ",
  "variables": {
    "id": "1234"
  }
}
```

Example Response

```
{
  "data": {
    "chirp": {
      "contents": "a chirp"
    }
  },
  "errors": null
}
```

GraphQL Schema Language

Primitives

`scalar` represent a single value. Predefined: ID, String, Int, Float, Boolean.

```
"""
Date and Time represented using the ISO 8601 standard
"""
scalar DateTime

"""
Email represented as a String
Format validated when used as input
"""
scalar Email
```

Primitives

`enum` represents an enumeration of possible values.

Serialized as strings.

```
"""
Cardinal directions
"""
enum Direction {
  NORTH
  EAST
  SOUTH
  WEST
}
```

Modifiers

`!` to denote a non-nullable field

```
"""
Non-nullable identifier
"""
id: ID!
```

Modifiers

`[]` to denote a list of elements, around the type

```
"""A nullable list of nullable strings"""
a: [String]

"""A non-nullable list of nullable strings"""
b: [String]!

"""A nullable list of non-nullable strings"""
c: [String!]

"""A non-nullable list of non-nullable strings"""
d: [String!]!
```

Objects

`input` used to represent a list of named and typed input fields, can only contain primitives and other input types

```
"""Input used to create a new chirp"""
input ChirpInput {
  """Contents of the chirp"""
  contents: String!
}
```

Objects

`type` used to represent a list of named and typed output fields

```
"""A Chirp"""
type Chirp {
  """Identifier of the chirp"""
  id: ID!

  """Contents of the chirp"""
  contents(format: ContentsFormat = PLAIN): String!
}
```

Abstractions

`interface` used to represent a list of named and typed output fields, implemented by types

```
"""An identified and queryable node within the graph"""
interface Node {
  """Unique non-nullable identifier of the node"""
  id: ID!
}

"""An identifiable Chirp"""
type Chirp implements Node {
  """Identifier of the chirp"""
  id: ID!

  """Contents of the chirp"""
  contents: String!
}
```

Abstractions

`union` to represent a collection of possible output types

```
"""A union of possible errors that can happen when creating a new chirp
union ChirpUsageError = EmptyContents | TooLongContents

"""Contents were empty"""
type EmptyContentsError implements UsageError {
  message: String!
}

"""Contents were too long"""
type TooLongContents implements UsageError {
  message: String!
  length: Int!
  maxLength: Int!
}
```

Directives

Annotations on the graph, prefixed with `@`, can have arguments. Predefined:

`@deprecated(reason: String!)`
: indicate to the client the element is deprecated, along with a reason

`@skip(if: Boolean!)`
: don't return the field's value when the condition is false

`@include(if: Boolean!)`
: return the field's value only when the condition is true

`@specifiedBy(url: String!)`
: add formatting information for scalars, usually pointed to a standard or specification

Directives

Custom directives can also be created

```
# defining a custom directive to validate authorization
# for output fields
directive @authorization(scopes: [String!]!) on FIELD_DEFINITION

type Mutation {
  chirp(input: ChirpInput!): ChirpPayload!
    @authorization(scopes: ["chirp.write"])
}
```

GraphQL Query Language

Description

Clients use the query language to request information from a GraphQL server.

```
query {
  chirp(id: "1234") {
    contents
    author { username }
  }
}
```

Variables

Variables can be passed to parameterize requests

```
query ($id: ID!) {
  chirp(id: $id) {
    contents
    author { username }
  }
}
```

```
{
  "query": "query($id){chirp(id:$id){contents author{username}}}",
  "variables": {
    "id": "1234"
  }
}
```

Resolver

Description

A resolver is a function that is used to return the value for a specific field within a type.

```
type User { name: String! }
```

```
resolvers = {
  User: {
    name: () => 'Sebastien Lavoie-Courchesne'
  }
}
```

Anatomy

Resolvers take in 4 parameters:

parent
: The parent object

arguments
: Any arguments passed to the field

context
: The context generated by the server

info
: Additional information about the request

Default Resolver

If no resolver is defined for a field:

```
function [Type.fieldName](
  parent: Type
): any {
  return parent[fieldName];
}
```

Parent

Whatever was returned by the resolver of the parent field, including properties that are not mapped to fields in the GraphQL schema

`null` if the parent type is Query, Mutation, or Subscription

Arguments

Any arguments passed to the field.

```
type Query {
  chirps(first: Int, after: String, last: Int, before: String):
    ChirpConnection!
}
```

```
interface ChirpsArguments {
  first: number;
  after: string;
  last: number;
  before: string;
}

function chirps(parent: null, arguments: ChirpsArguments) // ...
```

Context

A context object generated by the ApolloServer

```
new ApolloServer({
  context({ req, res }) {
    return {
      user: req.user,
    };
  },
  dataSources() {
    return {
      users: new UserDataSource(),
      chirps: new ChirpDataSource(),
    };
  }
});
```

⇒

```
{
  user,
  dataSources: {
    users,
    chirps,
  }
}
```

Data Sources

```
abstract class DataSource<TContext> {
  initialize?(config: DataSourceConfig<TContext>):
    void | Promise<void>;
}
```

Initialize is called on each incoming request

Instantiate a new data source on each incoming request

What to include?

Context:

- authentication information
- logger
- tracer + trace information

DataSources:

- repositories
- caches
- services
- clients to other services

info

```
interface GraphQLResolveInfo {
  fieldName: string;
  fieldNodes: FieldNode[];
  returnType: GraphQLOutputType;
  parentType: GraphQLObjectType;
  path: Path;
  schema: GraphQLSchema;
  fragments: Record<string, FragmentDefinitionNode>;
  rootValue: any;
  operation: OperationDefinitionNode;
  variableValues: Record<string, any>;
}
```

Scalars

Scalars can have additional parsing in the form of a 3 function object:

parseLiteral
: Used to parse values passed in through the query's text itself

parseValue
: Used to parse values passed through the variables

serialize
: Used to serialize the domain value to a String

These can be used to add validation (correct format, specific length, etc) as well as map it to another type (Date, URL, etc).

Directives

Directives can have additional logic tied to them in the form of a Visitor pattern.

Directives are not visible to the clients introspecting the schema.

These can be used to add validation (correct format, specific length, etc) or transformations (lower/uppercase, rounding, etc)

Federation

Specification

Apollo Federation is an additional specification to specify subgraphs, gateways and schema registries.

The gateway aggregates the subgraphs to expose a single GraphQL endpoint. Subgraphs can reference each other.

Subgraph

Individual service exposing a GraphQL endpoint. Services must satisfy an additional schema containing:

```
scalar _Any
scalar _FieldSet

# a union of all types that use the @key directive
union _Entity

type _Service {
  sdl: String
}

extend type Query {
  _entities(representations: [_Any!]!): [_Entity]!
  _service: _Service!
}
```

Subgraph directives

The following directives are available to allow referencing other types within other subgraphs:

```
directive @external on FIELD_DEFINITION
directive @requires(fields: _FieldSet!) on FIELD_DEFINITION
directive @provides(fields: _FieldSet!) on FIELD_DEFINITION
directive @key(fields: _FieldSet!) repeatable on OBJECT | INTERFACE

# not all implementations allow this one
# there's an "extend" keyword as well
directive @extends on OBJECT | INTERFACE
```

Subgraph References

Subgraphs can reference types from another subgraph by extending them:

```
extend type User @key(fields: "id") {
  id: ID! @external

  chirps: [Chirp!]!
}

type Chirp @key(fields: "id") {
  id: ID!

  author: User!
}
```

Returning references

Return the type and values for all the fields in the key:

```
{
  "__typename": "User",
  "id": "1234"
}
```

Resolving references

Special resolver to resolve an entity based on its key:

```
function __resolveReference(
  parent: { id: string },
  { dataSources: { users } }: Context
): Promise<User> {
  return users.getById(id);
}
```

Testing in isolation

`__resolveReference` can be tested by using the `_entities` query:

```
query {
  _entities(representations: [
    {
      __typename: "User",
      id: "1234"
    }
  ]) {
    ... on User {
      username
    }
  }
}
```

Gateway

Main responsabilities on server start and/or periodically:

- Validate schemas from each subgraph
- Aggregate schemas from each subgraph
- Expose single GraphQL endpoint

Main responsabilities on each request:

- Parse incoming requests
- Build query execution plan
- Execute plan, sending sub-requests to each subgraph
- Return aggregated result

Schema Registry

A Schema Registry can be used to validate and aggregate schemas.

The gateway then pulls the schema from the schema registry.

Best Practices

Federation

When working with federation across multiple teams:

- Define guidelines & standards early
- Have consistent naming strategies
- Review schemas to ensure they follow guidelines & standards

Node interface

Every identifiable type should implement this node interface

```
"""An identifiable node within the graph"""
interface Node {
  """Unique identifier of the node"""
  id: ID!
}
```

And the server can expose a node query to fetch any node within the graph

```
type Query {
  """
  A global node query to get any node within the
  graph by its unique identifier
  """
  node(id: ID!): Node
}
```

Connection Pattern

For paginated queries, use the connection patternRelay, *GraphQL Cursor Connections Specification*

```
type PageInfo {
  hasPreviousPage: Boolean!
  hasNextPage: Boolean!
  startCursor: String
  endCursor: String
}

type ChirpConnection {
  pageInfo: PageInfo!
  edges: [Chirp!]!
}

type Query {
  chirps(first: Int, after: String, last: Int, before: String):
    ChirpConnection!
}
```

Mutation Pattern

Have a consistent pattern for mutations

```
type Mutation {
  chirp(input: ChirpInput!): ChirpPayload!
}

input ChirpInput {
  contents: String!
}

type ChirpPayload {
  chirp: Chirp
  errors: [ChirpUsageError!]
}

union ChirpUsageError = EmptyContents | ContentsTooLong
```

Persisted Queries

Servers can allow creation of persisted queries.

Client sends static query text to server (separate endpoint)
Server parses, generates query plan, saves everything, return id

For each query, client sends id of the persisted query and variables

Rate Limiting

Rate limit by allocating points to each client

Calculate the cost of each request in points:

- querying a single node: 1 point
- querying a list: 1 point + 1 point per element
- executing a mutation: x points

Conclusion

Conclusion

Specifications


Apollo. *Apollo Federation Specification*.
https://www.apollographql.com/docs/federation/federation-spec/.


GraphQL-contributors. *GraphQL Specification, October 2021 Edition*.
https://spec.graphql.org/October2021/.


Relay. *GraphQL Cursor Connections Specification*.
https://relay.dev/graphql/connections.htm.

Material

 Lavoie-Courchesne, Sebastien. *GraphQL: Thinking in Resolvers*. https://lavoiecsh.github.io/technologies/2020/10/15/thinking-in-resolvers.html.

 — .*GraphQL: Thinking in Resolvers*. https://github.com/lavoiecsh/presentations/tree/main/confoo-2022/graphql-thinking-in-resolvers.