GraphQL:Thinking in Resolvers

Sébastien Lavoie-Courchesne

February 23rd 2022

opening

Sébastien Lavoie-Courchesne

- Architect for the Catalog Group at AppDirect
- Member of the GraphQL Working Group at AppDirect
- Developed most of the initial infrastructure we use

Contents

Technologies used in the examples

- GraphQL schemas
- ApolloServer stack:
  - NodeJS
  - Typescript
  - ApolloServer

What is GraphQL?

From the GraphQL Specification[1]:

> *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?

A specification defining a query language for communication between different services over a network and how to implement the server and client sides of the communication

Why GraphQL?

Only request and receive the fields you want:

- Smaller payloads
- Descriptive language
- Documented

Why GraphQL?

Only request and receive the fields you want:

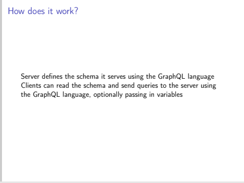
- Smaller payloads
- Descriptive language
- Documented

Often compared to REST and gRPC
Mostly used over HTTP, but not always

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



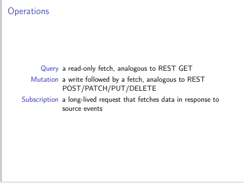
How does it work?

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

typically one query per request, multiple queries in a single request are supported

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



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

Subscription is less often used and less documented, multiple ways to serve it (WebSocket, Kafka, RabbitMQ, etc)

HTTP Communication

- Most common use of GraphQL
- HTTP GET or POST to a specific URL
- Traditional HTTP headers/cookies
- GET `query` parameter contains the query as a string
- POST Body includes

`query`	Query as a string
`variables`	Query variable values as a JSON object, optional
`operationName`	Name for the operation, optional

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 us
scalar Email
```

enum represent 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!
```

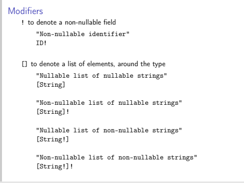
`[]` to denote a list of elements, around the type

```
"Nullable list of nullable strings"
[String]

"Non-nullable list of nullable strings"
[String]!

"Nullable list of non-nullable strings"
[String!]

"Non-nullable list of non-nullable strings"
[String!]!
```



Modifiers

- `!` to denote a non-nullable field

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

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

```
"Nullable list of nullable strings"
[String]

"Non-nullable list of nullable strings"
[String]!

"Nullable list of non-nullable strings"
[String!]

"Non-nullable list of non-nullable strings"
[String!]!
```

nullability can be used to indicate a missing value, returning null for a non-nullable type generates an error

an empty list is non-null

good practice to have non-nullable elements within the list

Objects

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

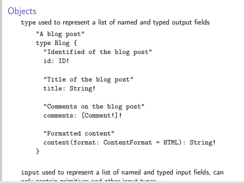
```
"A blog post"
type Blog {
  "Identified of the blog post"
  id: ID!

  "Title of the blog post"
  title: String!

  "Comments on the blog post"
  comments: [Comment!]!

  "Formatted content"
  content(format: ContentFormat = HTML): String!
}
```

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



Objects

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

```
"A blog post"
type Blog {
  "Identified of the blog post"
  id: ID!

  "Title of the blog post"
  title: String!

  "Comments on the blog post"
  comments: [Comment!]!

  "Formatted content"
  content(format: ContentFormat = HTML): String!
}
```

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

arguments can be added to fields to allow specifying more information about what should be returned, format, locale, page for collections, etc

only input objects and primitives can be used as arguments, input fields cannot have arguments

Abstractions

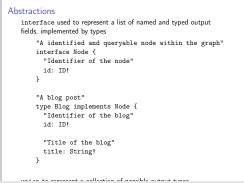
`interface` used to represent a list of named and typed output fields, implemented by types

```
"A identified and queryable node within the graph"
interface Node {
  "Identifier of the node"
  id: ID!
}

"A blog post"
type Blog implements Node {
  "Identifier of the blog"
  id: ID!

  "Title of the blog"
  title: String!
}
```

`union` to represent a collection of possible output types



Abstractions

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

```
"A identified and queryable node within the graph"
interface Node {
  "Identifier of the node"
  id: ID!
}

"A blog post"
type Blog implements Node {
  "Identifier of the blog"
  id: ID!

  "Title of the blog"
  title: String!
}
```

`union` to represent a collection of possible output types

any number of types can implement the interface

unions can be combined with interfaces to create more powerful abstractions, done by adding the same interface to every type in the union

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

```
# defining a custom directive to validate authorization
directive @authorization(scopes: [String!]!) on QUERY |

"A deprecated type"
type Comment @deprecated(reason: "Replaced by BlogComme

"Date and Time represented using the ISO 8601 standard"
scalar DateTime @specifiedBy(url: "https://www.iso.org/
```

Conclusion

Links

- blog
- this presentation

References

[1] *GraphQL Specification, October 2021 Edition*. GraphQL contributors. Oct. 2021. URL: https://spec.graphql.org/October2021/.