GraphQL
NodeJS
TypeScript
ApolloServer

Colourized frame on code examples
pink for GraphQL
blue for TS
black for JSON

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

Particularly useful when a server doesn't know about all its clients

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

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

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`

query as a string
variables as a JSON object
operationName is used to differentiate multiple requests sent in a single call on the client side

errors are returned within the response body, clients must parse it

Modifiers

! to denote a non-nullable field

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

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

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!]!
```

an empty list is non-null

good practice to have non-nullable elements within the list

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!
}
```

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

```
"""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!
}
```

any number of types can implement the interface

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!
}
```

unions can be combined with interfaces to create more powerful abstractions

done by adding the same interface to every type in the union

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"
  }
}
```

client-side advantage: static string construction (faster runtime)

server-side advantage: can cache query execution plan (faster responses)

If no resolver is defined for a field:

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

if field is a value, return it

if field is a function, execute it and return its output

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,
  }
}
```

executed on each request

contains the result of the context function as well as a dataSources property containing the result of the dataSources function

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

a second incoming request will re-initialize the data source

GraphQL:Thinking in Resolvers

Resolver

info

info

```ts
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>;
}
```

mostly extracted to other fields

path can be used to return the path when returning errors to identify where the error occurred

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!
}
```

this is entirely handled by most GraphQL implementations

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!
}
```

chirp service adds chirps field on the user type defined in the user service
chirp service also returns a reference to the user that authored the chirp
user service has no knowledge of chirps

Returning references

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

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

typename is only required when returning a union/interface

but a good practice to keep it

GraphQL:Thinking in Resolvers

Federation

Resolving references

parent, context, info

no arguments parameter

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
    }
  }
}
```

this is equivalent to the query sent by the gateway when it needs to resolve an entity by its key

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

The gateway then pulls the schema from the schema registry.

Apollo offers a paid online schema registry

there's also an open source schema registry that you can deploy yourself

Reduces load and potential failures on the gateway

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!
}
```

Originally designed by Relay

many client libraries understand this pattern and can automate the pagination

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
```

adding/removing fields is easier with an input object than multiple arguments

returning a payload with either the created/updated entity or a list of errors

also easier to update in the future

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

smaller payloads on each request because you don't send the full query text

faster execution on the server

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

multiple different algorithms to calculate the cost

exact points are up to you

can also specify points independently through directives