GraphQL

Edgio provides full support for caching GraphQL APIs. Putting Edgio in front of you GraphQL API can significantly improve its performance while reducing the amount of traffic that reaches your origin by serving cached queries from the network edge.

GraphQL History

GraphQL was built in 2012 to support Facebook mobile apps. Facebook open sourced the project in 2015, and in 2018, it was moved to the GraphQL Foundation.

What is GraphQL?

GraphQL is a specification that describes the behavior of a GraphQL server. It is a set of guidelines on how requests and responses should be handled like supported protocols, format of the data that can be accepted by the server, format of the response returned by the server, and so on.

GraphQL is not a graph database query language. You can use GraphQL to query data from any number of sources.

GraphQL is unopinionated about:

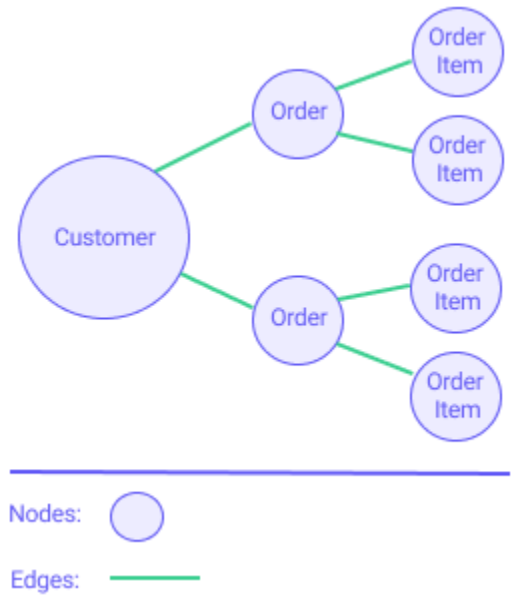
- The transport layer. It can be used with any available network protocol like TCP, websocket or any other transport layer protocol.
- Databases. You can use it with relational or NoSQL databases.
- Backend languages. Open source projects in a number of popular languages are available.

There are many open-source GraphQL servers that you can incorporate into your web application. We encourage you to investigate and choose your own.

Data Organization

Somewhat similar to graphs in mathematics and computer science, GraphQL's resources are nodes, and relations between resources are edges. GraphQL is organized as types and fields rather than endpoints.

All nodes extend from a root node. For example in an ordering system, a customer might have multiple orders, and each order would have one or more order items. In this case, customer, order, and order items are nodes and are connected by edges:



Benefits

GraphQL is strongly and statically typed, providing the following advantages:

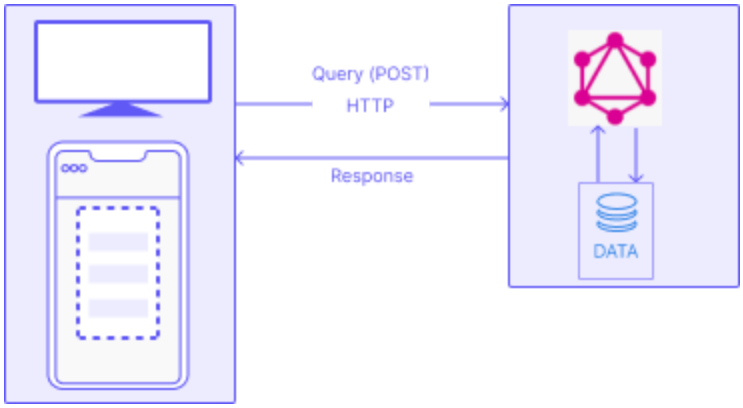
- Results are predictable, and queries are self-documenting.
- Because of its typing capabilities, GraphQL easily lends itself to code completion popups in an IDE, enhancing the developer experience.

GraphQL is efficient and yields the following benefits:

- It is designed to overcome the “not enough data returned (multiple round trips to the server)” and “too much data returned (n+1)” problems that often accompany REST APIs. If you use a REST API, you often have to make multiple requests to get the data you want. Also REST APIs often return more data than you need, increasing bandwidth, response time, and forcing you to parse large datasets for the desired content.
- GraphQL reduces the need for client-side error-handling and retry logic.

Architecture

A GraphQL client device makes HTTP POST requests to the GraphQL server.



The preceding figure shows a simplistic architecture. In reality other designs are possible, such as integrating a server with existing legacy systems.

When a GraphQL server receives a query, the server parses the requested payload and returns needed data.

GraphQL Operations

GraphQL supports the following operations:

- Query operations are read-only, similar to GET operations in REST.
- Mutation operations create, modify, and delete data; similar to POST, PUT, and DELETE in REST.
- Subscriptions observe event(s) and send data when an event occurs. With a subscription, your client application keeps a connection to the GraphQL server open. Subscriptions are generally used to notify your client in real time about changes to back-end data, such as the creation of a new object or updates to an important field.

Query and mutation operations also extend from a root node.

GraphQL Types

GraphQL supports the following built-in scalar types:

- Int: A signed 32-bit integer.
- Float: A signed double-precision floating-point value.
- String: A UTF-8 character sequence.
- Boolean: true or false.
- ID: A unique identifier, often used to refetch an object or as the key for a cache.

GraphQL also supports programmer-defined objects, usually data resources.

NOTE: Query and Mutation operations are types.

GraphQL Schemas

Schemas define your application's resources, the relationships between resources, and the operations that are allowed on the resources. Resources and operations are types. Schemas are also used to validate queries and mutations on your data. If the query or mutation structure matches the schema, the operation is executed.

For example, in an app where you maintain customers, you would have a Customer type and query operation types such as getAllCustomersand getCustomerById. You would also have mutation types like createCustomer and deleteCustomer. Operation types are somewhat like function prototypes; they simply define input and output. The function implementations are defined in Resolvers.

Resolvers

Resolvers are the actual functions that perform the operations defined in the schema and contain procedural code. There is a one-to-one relationship between operations and resolvers.

Schema and Resolver Examples

The examples in this section are modeled using Express (a Node.js web framework) and JavaScript.

Schemas

Going back to our Customer example, let's say that each customer has a name, address, city, country, and id. You could model the Customer resource and related operations as follows.

```
type Customer {
  name: String!
    address: String!
    city: String!
    country: String!
    id: ID!
}
#Get all customers
type Query {
    getAllCustomers: [Customer!]!
}

#Get one customer
type Query {
    getCustomerById(id: ID!): Customer!
}
```

Notice the following about this example:

- The exclamation point ! is the “required” or “not null” operator and indicates that an attribute is required when submitting a request, and is guaranteed to be returned by the server.
- Parameters are enclosed in parentheses; operations with no parameters lack parentheses.
- The colon following a function name indicates the return type.
- Square brackets [] are array notation and indicate a list. In the example, getAllCustomers returns a list of Customer resources.
- The Customer resource and its operations are both defined with the type keyword, signifying that they are types.

Resolvers

Using the Customer example, the resolvers might look like this:

```
type Customer {
  name: String!
    address: String!
    city: String!
    country: String!
    id: ID!
}

#Get all customers
type Query {
    getAllCustomers: [Customer!]!
}

#Get one customer
type Query {
    getCustomerById(id: ID!): Customer!
}
```

Notice the following about this example:

- The function names in the resolvers match the names in the schemas.
- The variable custs is an array of Customer resources. We’re using it instead of say, database query results for simplicity.
- The obj parameter is for a more advanced discussion of GraphQL.

Query Structure

Queries can be quite complex with multiple nested queries and variables, but we will keep our example simple.

Queries begin with the query keyword followed by a resource name, followed by a list of fields you want to see.

Query with No Parameters

Assume that you want to get the name and address of all customers. Your query would look like this:

```
getAllCustomers {
    name
    address
}
```

Notice the following about this example:

- The query is self-documenting, clearing showing that you want to retrieve all customers and that you want only name and address.
- Parentheses are not required after getAllCustomers.
- The list of fields is customizable; you include just the fields you want.

Query with a Parameter

Now assume that you want to get the name and address of the customer with id 01224950. Your query would look like this:

```
getCustomerById(id: "2210194") {
  name
  address
}
```

Notice the following about this example:

- The parameter name id matches the parameter name on the Customer type.
- Again, the list of fields is customizable; you include just the fields you want.

Additional GraphQL Capabilities

This topic has presented a high-level view of GraphQL. GraphQL contains many more capabilities among which are the use of the following:

- aliases
- fragments
- variables
- interfaces
- enums

- unions
- input fields

See the GraphQL Learning Web Site website for more information.