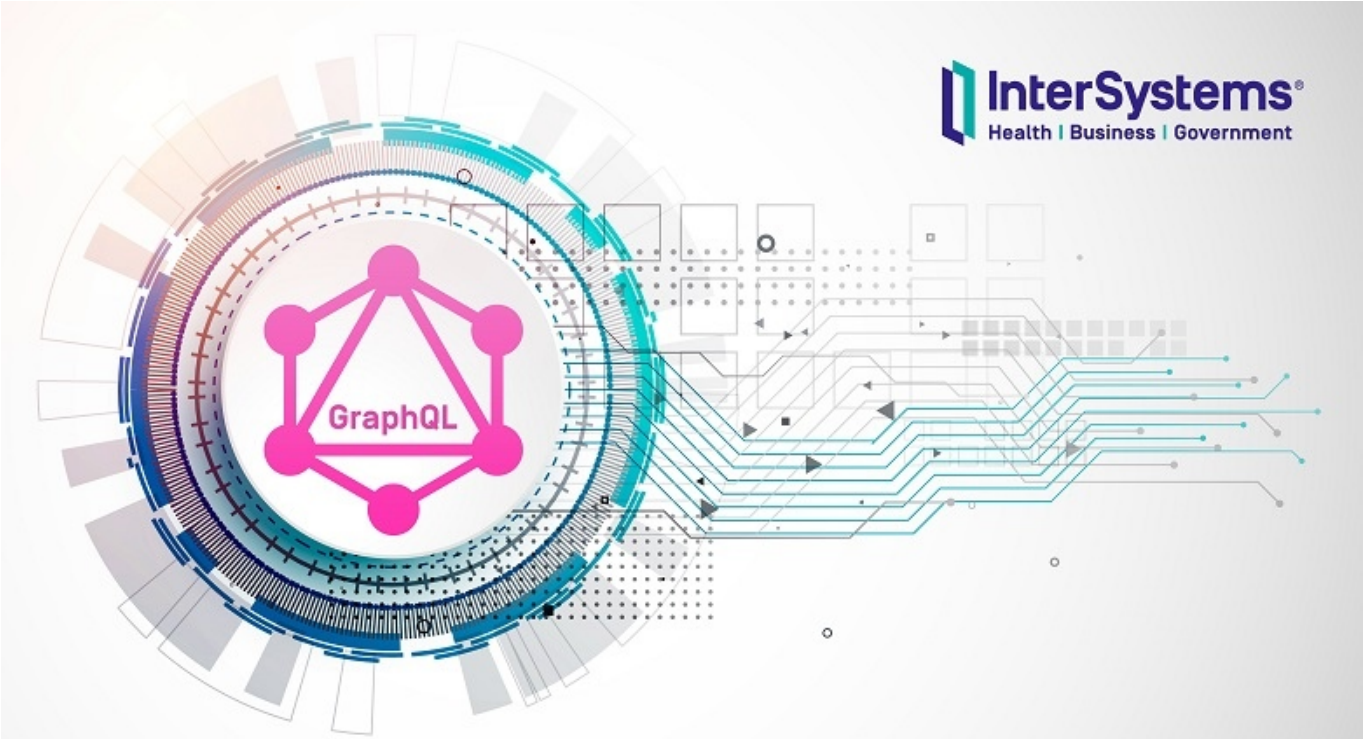
Article
Gevorg Arutiunian · Sep 4, 2018 11m read

How I implemented GraphQL for InterSystems platforms



I already talked about GraphQL and the ways of using it in this article. Now I am going to tell you about the tasks I was facing and the results that I managed to achieve in the process of implementing GraphQL for InterSystems platforms.

What this article is about

- Generation of an AST for a GraphQL request and its validation
- Generation of documentation
- Generation of a response in the JSON format

Let ' s take a look at the entire process from sending a request to receiving a response using this simple flow as an example:

The client can send requests of two types to the server:

- A schema request.
  The server generates a schema and returns it to the client. We ' ll cover this process later in this article.
- A request to fetch/modify a particular data set.
  In this case, the server generates an AST, validates and returns a response.

AST generation

The first task that we had to solve was to parse the received GraphQL request. Initially, I wanted to find an external library and use it to parse the response and generate an AST. However, I discarded the idea for a number of

reasons. This is yet another black box, and you should keep the issue with long callbacks in mind.

That ' s how I ended up with a decision to write my own parser, but where do I get its description? Things got better when I realized that GraphQL was an open-source project with a pretty good description by Facebook. I also found multiple examples of parsers written in other languages.

You can find a description of an AST here.

Let ' s take a look at a sample request and tree:

```
{
  Sample_Company(id: 15) {
    Name
  }
}
```

AST

```
{
  "Kind": "Document",
  "Location": {
    "Start": 1,
    "End": 45
  },
  "Definitions": [
    {
      "Kind": "OperationDefinition",
      "Location": {
        "Start": 1,
        "End": 45
      },
      "Directives": [],
      "VariableDefinitions": [],
      "Name": null,
      "Operation": "Query",
      "SelectionSet": {
        "Kind": "SelectionSet",
        "Location": {
          "Start": 1,
          "End": 45
        },
        "Selections": [
          {
            "Kind": "FieldSelection",
            "Location": {
              "Start": 5,
              "End": 44
            },
            "Name": {
              "Kind": "Name",
              "Location": {
                "Start": 5,
                "End": 20
              },
              "Value": "Sample_Company"
            },
```

```
          "Alias": null,
          "Arguments": [
            {
              "Kind": "Argument",
              "Location": {
                "Start": 26,
                "End": 27
              },
              "Name": {
                "Kind": "Name",
                "Location": {
                  "Start": 20,
                  "End": 23
                },
                "Value": "id"
              },
              "Value": {
                "Kind": "ScalarValue",
                "Location": {
                  "Start": 24,
                  "End": 27
                },
                "KindField": 11,
                "Value": 15
              }
            }
          ],
          "Directives": [],
          "SelectionSet": {
            "Kind": "SelectionSet",
            "Location": {
              "Start": 28,
              "End": 44
            },
            "Selections": [
              {
                "Kind": "FieldSelection",
                "Location": {
                  "Start": 34,
                  "End": 42
                },
                "Name": {
                  "Kind": "Name",
                  "Location": {
                    "Start": 34,
                    "End": 42
                  },
                  "Value": "Name"
                },
                "Alias": null,
                "Arguments": [],
                "Directives": [],
                "SelectionSet": null
              }
            ]
          }
        }
      ]
    }
```

```
    }
  ]
}
```

Validation

Once we receive a tree, we ' ll need to check if it has classes, properties, arguments and their types on the server – that is, we ' ll need to validate it. Let ' s traverse the tree recursively and check whether its elements match the ones on the server. Here ' s how a class looks.

Schema generation

A **schema** is a type of documentation for available classes and properties, as well as a description of property types in these classes.

GraphQL implementations in other languages or technologies use resolvers to generate schemas. A resolver is a description of the types of data available on the server.

Examples or resolvers, requests, and responses

```
type Query {
  human(id: ID!): Human
}

type Human {
  name: String
  appearsIn: [Episode]
  starships: [Starship]
}

enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}

type Starship {
  name: String
}
```

```
{
  human(id: 1002) {
    name
    appearsIn
    starships {
      name
    }
  }
}
```

```
{
  "data": {
    "human": {
```

```
      "name": "Han Solo",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "starships": [
        {
          "name": "Millenium Falcon"
        },
        {
          "name": "Imperial shuttle"
        }
      ]
    }
  }
}
```

However, before we generate a schema, we need to understand its structure, find a description or, even better, examples. The first thing I tried was attempting to find one that would help me understand the structure of a schema. Since GitHub has its own GraphQL API, it was easy to get one from there. But the problem was that its server-side was so huge that the schema itself occupied 64 thousand lines. I really hated the idea of delving into all that and started looking for other methods of obtaining a schema.

Since our platforms are based on a DBMS, my plan for the next step was to build and start GraphQL for PostgreSQL and SQLite. With PostgreSQL, I managed to fit the schema into just 22 thousand lines, and SQLite gave me an even better result with 18 thousand lines. It was better than the starting point, but still not enough, so I kept on looking.

I ended up choosing a NodeJS implementation, made a build, wrote a simple resolver, and got a solution with just 1800 lines, which was way better!

Once I had wrapped my head around the schema, I decided to generate it automatically without creating resolvers on the server in advance, since getting meta information about classes and their relationships is really easy.

To generate your own schema, you need to understand a few simple things:

- You don ’ t need to generate it from scratch – take one from NodeJS, remove the unnecessary stuff and add the things that you do need.
- The root of the schema has a queryType type. You need to initialize its “ name ” field with some value. We are not interested in the other two types since they are still being implemented at this point.
- You need to add all the available classes and their properties to the **types** array.

```
{
"data": {
    "__schema": {
        "queryType": {
            "name": "Query"
        },
        "mutationType": null,
        "subscriptionType": null,
        "types":[...
        ],
        "directives":[...
        ]
    }
}
```

```
}
```

- First of all, you need to describe the **Query** root element and add all the classes, their arguments, and class types to the **fields** array. This way, they will be accessible from the root element.

Let ' s take a look at two sample classes, ExampleCity and ExampleCountry

```
{
    "kind": "OBJECT",
    "name": "Query",
    "description": "The query root of InterSystems GraphQL interface.",
    "fields": [
        {
            "name": "Example_City",
            "description": null,
            "args": [
                {
                    "name": "id",
                    "description": "ID of the object",
                    "type": {
                        "kind": "SCALAR",
                        "name": "ID",
                        "ofType": null
                    },
                    "defaultValue": null
                },
                {
                    "name": "Name",
                    "description": "",
                    "type": {
                        "kind": "SCALAR",
                        "name": "String",
                        "ofType": null
                    },
                    "defaultValue": null
                }
            ],
            "type": {
                "kind": "LIST",
                "name": null,
                "ofType": {
                    "kind": "OBJECT",
                    "name": "Example_City",
                    "ofType": null
                }
            },
            "isDeprecated": false,
            "deprecationReason": null
        },
        {
            "name": "Example_Country",
            "description": null,
            "args": [
                {
                    "name": "id",
                    "description": "ID of the object",
                    "type": {
```

```
                        "kind": "SCALAR",
                        "name": "ID",
                        "ofType": null
                    },
                    "defaultValue": null
                },
                {
                    "name": "Name",
                    "description": "",
                    "type": {
                        "kind": "SCALAR",
                        "name": "String",
                        "ofType": null
                    },
                    "defaultValue": null
                }
            ],
            "type": {
                "kind": "LIST",
                "name": null,
                "ofType": {
                    "kind": "OBJECT",
                    "name": "Example_Country",
                    "ofType": null
                }
            },
            "isDeprecated": false,
            "deprecationReason": null
        }
    ],
    "inputFields": null,
    "interfaces": [],
    "enumValues": null,
    "possibleTypes": null
}
```


- Our second step is to go one level higher and extend the **types** array with the classes that have already been described in the **Query** object with all of the properties, types, and relationships with other classes.

Descriptions of classes


```
 {
"kind": "OBJECT",
"name": "Example_City",
"description": "",
"fields": [
    {
        "name": "id",
        "description": "ID of the object",
        "args": [],
        "type": {
            "kind": "SCALAR",
            "name": "ID",
            "ofType": null
        },
        "isDeprecated": false,
        "deprecationReason": null
```

```
    },
    {
        "name": "Country",
        "description": "",
        "args": [],
        "type": {
            "kind": "OBJECT",
            "name": "Example_Country",
            "ofType": null
        },
        "isDeprecated": false,
        "deprecationReason": null
    },
    {
        "name": "Name",
        "description": "",
        "args": [],
        "type": {
            "kind": "SCALAR",
            "name": "String",
            "ofType": null
        },
        "isDeprecated": false,
        "deprecationReason": null
    }
],
"inputFields": null,
"interfaces": [],
"enumValues": null,
"possibleTypes": null
},
{
"kind": "OBJECT",
"name": "Example_Country",
"description": "",
"fields": [
    {
        "name": "id",
        "description": "ID of the object",
        "args": [],
        "type": {
            "kind": "SCALAR",
            "name": "ID",
            "ofType": null
        },
        "isDeprecated": false,
        "deprecationReason": null
    },
    {
        "name": "City",
        "description": "",
        "args": [],
        "type": {
            "kind": "LIST",
            "name": null,
            "ofType": {
                "kind": "OBJECT",
                "name": "Example_City",
                "ofType": null
```

```
            }
        },
        "isDeprecated": false,
        "deprecationReason": null
    },
    {
        "name": "Name",
        "description": "",
        "args": [],
        "type": {
            "kind": "SCALAR",
            "name": "String",
            "ofType": null
        },
        "isDeprecated": false,
        "deprecationReason": null
    }
],
"inputFields": null,
"interfaces": [],
"enumValues": null,
"possibleTypes": null
}
```

- The third point is that the “ types ” array contains descriptions of all popular scalar types, such as int, string, etc. We ’ ll add our own scalar types there, too.

Response generation

Here is the most complex and exciting part. A request should generate a response. At the same time, the response should be in the JSON format and match the request structure.

For each new GraphQL request, the server has to generate a class containing the logic for obtaining the requested data. The class is not considered new if the values of arguments changed – that is, if we get a particular dataset for Moscow and then the same set for London, no new class will be generated, it ’ s just going to be new values. In the end, this class will contain an SQL query and the resulting dataset will be saved in the JSON format with its structure matching the GraphQL request.

An example of a request and a generated class

```
{
  Sample_Company(id: 15) {
    Name
  }
}
```

```
Class gqlcq.qsmytrXzYZmD4dvgwVIIA [ Not ProcedureBlock ]
{

ClassMethod Execute(arg1) As %DynamicObject
{
    set result = {"data":{}}
    set query1 = []

    #SQLCOMPILE SELECT=ODBC
```

```
    &sql(DECLARE C1 CURSOR FOR
         SELECT  Name
         INTO :f1
         FROM Sample.Company
         WHERE id= :arg1
)   &sql(OPEN C1)
    &sql(FETCH C1)
    While (SQLCODE = 0) {
        do query1.%Push({"Name":(f1)})
        &sql(FETCH C1)
    }
    &sql(CLOSE C1)
    set result.data."Sample_Company" = query1

    quit result
}

ClassMethod IsUpToDate() As %Boolean
{
   quit:$$$comClassKeyGet("Sample.Company",$$$cCLASShash)'="3B5DBWmwgoE" $$$NO
   quit $$$YES
}
}
```

How this process looks in a scheme:

For now, the response is generated for the following requests:

- Basic
- Embedded objects
    - Only the many-to-one relationship
- List of simple types
- List of objects

Below is a scheme containing the types of relationships that are yet to be implemented:

Summary

- **Response** — at the moment, you can get a set of data for relatively simple requests.
- **Automatically generated schema** — the schema is generated for stored classes accessible to the client, but not for pre-defined resolvers.
- **A fully functional parser** – the parser is fully implemented, you can get a tree by making a request of any complexity.

→ Link to the project repository
→ Link to the demo server

#API #REST API #Caché #Ensemble #HealthShare #InterSystems IRIS

Source URL:https://community.intersystems.com/post/how-i-implemented-graphql-intersystems-platforms