GraphQL for Delphi introduction

What is GraphQL?

GraphQL is a modern way to build HTTP APIs consumed by the web and mobile clients. It is intended to be an alternative to REST and SOAP APIs (even for existing applications).

From GraphQL itself:

> GraphQL is a query language for your API, and a server-side runtime for executing queries by using a type system you define for your data. GraphQL isn't tied to any specific database or storage engine and is instead backed by your existing code and data.

GraphQL provides a standard way to:

- describe data provided by a server in a statically typed Schema;
- request data in a Query which exactly describes your data requirements; and
- receive data in a Response containing only the data you requested.

GraphQL was developed internally by Facebook in 2012 before being publicly released in 2015. Learn more about GraphQL language at https://graphql.org/learn/.

About GraphQL for Delphi

GraphQL for Delphi is an implementation of GraphQL specification for Delphi.

GraphQL request is validated and executed by GraphQL server with the schema of the GraphQL service and a GraphQL document which contains the operations.

GraphQL for Delphi supports Delphi 10 Seattle and later versions up to the most recent one.

Features

Below is a list of exists GraphQL for Delphi features and GraphQL specification elements covered.

- **Full GraphQL document parser with support to:**
  - Full spec-compliant document lexer;
  - Executable documents and type definition documents;
  - Operations;
  - Selection sets;
  - Fields;
  - Arguments;
  - Field aliases;
  - Fragments (including inline and type conditions);
  - Input values;

    - All types supported (Int, String, Object, etc.);
    - Variables;
    - Type references (List, Non-Null);
    - Directives.
- **GraphQL schema supporting the following types:**
    - Query and Mutation root types;
    - Scalars: Int, Float, String, Boolean and ID;
    - Objects, field arguments and field deprecation;
    - Interfaces;
    - Unions;
    - Enums;
    - Input objects;
    - List types;
    - Non-null types;
    - Directives, including @skip, @include and @deprecated.
- **Full Introspection support;**
- **Full Validation support;**
- **Spec-compliant GraphQL document execution** (*)
    - Execute a GraphQL document based on a schema and retrieve results;
    - Execution strictly following GraphQL specification;
    - Skip/include directives handling;
    - Fragments;
    - Selection set execution;
    - Variables;
    - Fields and variables values coercion;
    - Field resolvers and abstract type resolvers;
    - Proper error handling in response with precise error location and extensions;
    - Automatic field resolver binding using RTTI.
- **GraphQL over HTTP:**
    - GraphQL HTTP handler compliant to upcoming GraphQL over HTTP specification;
    - WebBroker dispatcher component:
        - Support for Windows and Linux servers (**);
        - Deploy with Apache, IIS, FastCGI, Standalone (***);
    - Automatic JSON serialization/deserialization.
- **GraphQL Playground IDE built-in:**
    - Execute queries and debug your GraphQL server from web easily by enabling GraphQL Playground;
    - Fully customizable (light/dark theme, title, etc.).
- **Fully extensible:**
    - Abstract HTTP request/response allows using the GraphQL HTTP handler with any Delphi HTTP framework;
    - Inheritable schema types allows creating your GraphQL types, including scalars.
- **Extensive documentation including full API reference.**
- **Supports from Delphi 10 Seattle up to the latest available Delphi version.**
- **Platforms support: Windows, Linux, macOS, Android and iOS.**
- **Premium support.**

(*) Unsupported features still under work: type extensions and subscriptions.

(**) Linux support requires Delphi Enterprise with Linux compiler.

(***) Functionality provided by WebBroker technology itself.

Documentation Topics

Please refer to the following topics in this documentation. This guide is also available as PDF at graphql-user-guide.pdf and as Microsoft Compiled HTML Help at graphqldoc.chm.

In this section:

Getting started

Getting started with GraphQL for Delphi.

Schema

Understanding GraphQL schema and creating a schema document.

Release Notes

List of releases, new features added and bugs fixed.

Breaking Changes

Information about breaking changes introduced by released.

Copyright Notice

Copyright information about this library.

Getting started

In this article, we will walk you through the basics of installing GraphQL for Delphi and creating a GraphQL server with it.

Installation

You can use GraphQL for Delphi starting in Delphi 10 Seattle and up to the latest Delphi versions. GraphQL for Delphi is distributed and supported by TMS Software.

You can refer to the Download page to find the links to download installer for GraphQL for Delphi for your specified Delphi version:

GraphQL for Delphi Download Page

Installation is pretty straightforward, just download and run the installer.

GraphQL for Delphi is also available in GetIt. Go to Delphi menu option `Tools | GetIt Package Manager...` and search for `graphql`. Select GraphQL for Delphi from the list of available filtered packages and click "Install".

Creating a Hello World GraphQL server

Once you have installed GraphQL for Delphi, you can build your first Hello World GraphQL server, using the following tutorial.

1. Create a new WebBroker-based web application

- Choose Delphi menu option `File -> New -> Other...`;
- Select `Delphi -> Web` category, and then double click option `Web Server Application`;
- Follow the wizard, choosing `Stand-alone GUI Application` as WebBroker project type and `VCL application` as application type;
- Once finished, the wizard will create a project with two units: `FormUnit1` and `WebModuleUnit1`.

2. Define your GraphQL schema

- Open data module `WebModule1` in unit `WebModuleUnit1`;
- Drop the component TGraphQLSchema in the data module.
- Using object inspector, edit the Definition property and paste the following text:

```
type Query {
     hello(name: String!): String
}
```

3. Setting up the field resolver

- Still in the object inspector, double click the OnInitSchema event;
- Paste the following implementation in the new created event handler:

```
procedure TWebModule1.GraphQLSchema1InitSchema(Sender: TObject; Schema: TSchemaDo
cument);
begin
  Schema.SetResolver('Query', 'hello', function(Args: TFieldResolverArgs): TValue
    begin
      Result := 'Hello, ' + Args.GetArgument('name').AsString;
    end
  );
end;
```

4. Adding the WebBroker dispatcher

- Drop the component TGraphQLWebBrokerDispatcher in the data module;
- Double click the Schema property of the component to associate the dispatcher to the previously added TGraphQLSchema component;
- Set property Options.PlaygroundEnabled to `True`.

5. Running and testing the GraphQL server

Your GraphQL server is now ready to run.

- Run the server, and then click the `Open Browser` function (or simply use your web browser to navigate to url `http://localhost:8080`.
- The browser should open the GraphQL Playground IDE. Using the editor at the left side, type the following query:

```
query {
  hello(name: "World")
}
```

- Click the `Play` button in the middle of the screen, and you should get the server response at the right editor:

```
{
  "data": {
    "hello": "Hello, World"
  }
}
```

Congratulations! You have built your first GraphQL server using Delphi!

Bookshelf server: a more complex example

Let's create a sample Bookshelf application to put all pieces together.

1. Create a new Web Server Application using `File -> New -> Other...` dialog.

Choose `Stand-alone GUI Application` as WebBroker project type and `VCL application` as application type.

2. Create new unit GraphQL.Bookshelf.pas with resolvers implementation.

For demo purposes we are not using any database, only simple `TList<T>` instances. Use the following code as the full unit source code.

```
unit GraphQL.Bookshelf;

interface

uses
  Generics.Collections,
  GraphQL.Main,
  GraphQL.Schema;

type
  TBook = class;

  TAuthor = class
  private
    FId: Integer;
    FName: string;
    function GetBooks: TArray<TBook>;
  public
    constructor Create(AName: string);
    property Books: TArray<TBook> read GetBooks;

    property Id: Integer read FId;
    property Name: string read FName;
  end;

  TBook = class
  private
    FId: Integer;
    FName: string;
    FAuthorId: Integer;
    function GetAuthor: TAuthor;
  public
    constructor Create(AName: string; AAuthorID: Integer);
    property Id: Integer read FId;
    property Name: string read FName;
    property Author: TAuthor read GetAuthor;
  end;
```

```
  TMutation = class
  public
    function CreateAuthor(Name: string): TAuthor;
    function CreateBook(Name: string; AuthorID: Integer): TBook;
  end;

  TQuery = class
    function Books: TArray<TBook>;
    function Authors: TArray<TAuthor>;
    Function Book(id: Integer): TBook;
    function Author(id: Integer): TAuthor;
  end;

implementation

var
  Counter: Integer;
  BookList: TObjectList<TBook>;
  AuthorList: TObjectList<TAuthor>;

{ TBook }

constructor TBook.Create(AName: string; AAuthorID: Integer);
begin
  inherited Create;
  FName := AName;
  FAuthorID := AAuthorID;
  FId := Counter;
  Inc(Counter);
end;

function TBook.GetAuthor: TAuthor;
var
  I: Integer;
begin
  Result := nil;
  for I := 0 to AuthorList.Count - 1 do
    if AuthorList[I].Id = Self.FAuthorId then
    begin
      Result := AuthorList[I];
      Break;
    end;
end;

{ TMutation }

function TMutation.CreateAuthor(Name: string): TAuthor;
begin
  Result := TAuthor.Create(Name);
  try
    AuthorList.Add(Result)
  except
```

```
    Result.Free;
    raise;
  end;
end;

function TMutation.CreateBook(Name: string; AuthorID: Integer): TBook;
begin
  Result := TBook.Create(Name, AuthorID);
  try
    BookList.Add(Result)
  except
    Result.Free;
    raise;
  end;
end;

{ TAuthor }

constructor TAuthor.Create(AName: string);
begin
  inherited Create;
  FName := AName;
  FId := Counter;
  Inc(Counter);
end;

function TAuthor.GetBooks: TArray<TBook>;
var
  I: Integer;
  Books: TList<TBook>;
begin
  Books := TList<TBook>.Create;
  try
    for I := 0 to BookList.Count - 1 do
      if BookList[I].FAuthorId = Self.Id then
        Books.Add(BookList[I]);

    Result := Books.ToArray;
  finally
    Books.Free;
  end;
end;

{ TQuery }

function TQuery.Author(id: Integer): TAuthor;
var
  I: Integer;
begin
  Result := nil;
  for I := 0 to AuthorList.Count - 1 do
    if AuthorList[I].Id = id then
```

```
    begin
      Result := AuthorList[I];
      Break;
    end;
end;

function TQuery.Authors: TArray<TAuthor>;
begin
  Result := AuthorList.ToArray;
end;

function TQuery.Book(id: Integer): TBook;
var
  I: Integer;
begin
  Result := nil;
  for I := 0 to BookList.Count - 1 do
    if BookList[I].Id = id then
    begin
      Result := BookList[I];
      Break;
    end;
end;

function TQuery.Books: TArray<TBook>;
begin
  Result := BookList.ToArray;
end;

initialization
  Counter := 1;
  BookList := TObjectList<TBook>.Create(True);
  AuthorList := TObjectList<TAuthor>.Create(True);

finalization
  BookList.Free;
  AuthorList.Free;

end.
```

3. Configure the GraphQL Server application

- Add GraphQL.Bookshelf to WebModuleUnit1 uses.
- Open WebModuleUnit1 designer and drop TGraphQLSchema and TGraphQLWebBroker Dispatcher components there.
- In `GraphQLWebBrokerDispatcher1` properties set `Schema` to our newly created `GraphQLSchema1` and `Options.PlaygroundEnabled` to `True`.
- Copy full GraphQL schema definition into GraphQLSchema1.Definition property:

```
type Book {
  id: ID!
  name: String!
  author: Author!
}

type Author {
  id: ID!
  name: String!
  books: [Book!]!
}

type Query {
  author(id: ID!): Author
  book(id: ID!): Book
  books: [Book!]!
  authors: [Author!]!
}

type Mutation {
  createAuthor(name: String!): Author!
  createBook(name: String!, authorId: ID!): Book!
}
```

- Set bindings between resolvers and GraphQL schema in TGraphQLSchema.OnInitSchema handler:

```
procedure TWebModule1.GraphQLSchema1InitSchema(Sender: TObject;
  Schema: TSchemaDocument);
begin
  Schema.Bind('Query', TQuery);
  Schema.Bind('Mutation', TMutation);
end;
```

4. Run and test the server

Now the GraphQL Server application is ready to use. Start the application, click Start button and open `http://localhost:8080/graphql/` in your browser. You should see GraphQL Playground IDE and now you can perform queries to your GraphQL server.

Here are some query examples you can use:

```
# create author, return author id
mutation {
  createAuthor(name: "Leo") {
    id
  }
}

# create book, return book and author
mutation {
  createBook(name: "War and Peace", authorId: 1) {
    id
    author {
      name
    }
  }
}

# return all books with authors
query {
  books {
    name
    author {
      id
      name
    }
  }
}

# get author books using variables
query GetAuthor($id: ID!) {
  author(id: $id) {
    name
    books {
      id
      name
    }
  }
}
# variables:
{
  "id": 1
}
```

GraphQL Schema

Your GraphQL server uses a schema to describe the shape of your available data. This schema defines a hierarchy of types with fields that are populated from your back-end data stores. The schema also specifies exactly which queries and mutations are available for clients to execute.

This article describes the fundamental building blocks of a schema and how to create a schema using GraphQL for Delphi.

The schema definition language

The GraphQL specification defines a human-readable schema definition language (or SDL) that you use to define your schema and store it as a string.

Here's a short example schema that defines two object types: `Book` and `Author`:

```
type Book {
  id: ID
  name: String
  author: Author
}

type Author {
  id: ID
  name: String
  books: [Book]
}
```

A schema defines a collection of types and the relationships between those types. In the example schema above, a `Book` can have an associated author, and an `Author` can have a list of books.

Because these relationships are defined in a unified schema, client developers can see exactly what data is available and then request a specific subset of that data with a single optimized query.

Note that the schema is not responsible for defining where data comes from or how it's stored. It is entirely implementation-agnostic.

Field definitions

Most of the schema types you define have one or more fields:

```
# This Book type has two fields: name and author
type Book {
  id: ID          # record ID
  name: String    # returns a String
  author: Author # returns an Author
}
```

A field can return a list containing items of a particular type. You indicate list fields with square brackets [], like so:

```
    type Author {
      id: ID
      name: String
      books: [Book] # A list of Books
    }
```

By default, it's valid for any field in your schema to return `null` instead of its specified type. You can require that a particular field doesn't return `null` with an exclamation mark `!`, like so:

```
    type Author {
      id: ID!       # Can't return null
      name: String! # Can't return null
      books: [Book]
    }
```

These fields are non-nullable. If your server attempts to return null for a non-nullable field, an error is thrown.

With a list field, an exclamation mark `!` can appear in any combination of two locations:

```
    type Author {
      id: ID!         # Can't return null
      name: String!   # Can't return null
      books: [Book!]! # This list can't be null AND its list *items* can't be
null
    }
```

- If `!` appears inside the square brackets, the returned list can't include items that are null.
- If `!` appears outside the square brackets, the list itself can't be null.
- In any case, it's valid for a list field to return an empty list.

The `Query` type

The `Query` type is a special object type that defines all of the top-level **entry points** for queries that clients execute against your server.

Each field of the `Query` type defines the name and return type of a different entry point. The `Query` type for our example schema might resemble the following:

```
    type Query {
      author(id: ID!): Author
      book(id: ID!): Book
      books: [Book!]!
      authors: [Author!]!
    }
```

This `Query` type defines four fields. `author` and `book` fields return a record of the corresponding type by its `id`, `books` and `authors` fields return a list of the corresponding type.

With a REST-based API, this would probably be returned by different endpoints (e.g., `/api/books`, `/api/authors`, `/api/books/:id`, `/api/authors/:id`). The flexibility of GraphQL enables clients to query both resources with a single request.

Structuring a query

Based on our example schema so far, a client could execute the following query, which requests both a list of all book names *and* a list of all author names:

```
query GetBooksAndAuthors {
  books {
    name
  }

  authors {
    name
  }
}
```

Our server would then respond to the query with results that match the query's structure, like so:

```
{
  "data": {
    "books": [
      {
        "name": "War and Peace"
      },
      ...
    ],
    "authors": [
      {
        "name": "Leo Tolstoy"
      },
      ...
    ]
  }
}
```

The `Mutation` type

The `Mutation` type is similar in structure and purpose to the `Query` type. Whereas the `Query` type defines entry points for *read* operations, the `Mutation` type defines entry points for *write* operations.

Each field of the `Mutation` type defines the signature and return type of a different entry point. The `Mutation` type for our example schema might resemble the following:

```
type Mutation {
  createAuthor(name: String!): Author!
  createBook(name: String!, authorId: ID!): Book!
}
```

This `Mutation` type defines available mutations, `createAuthor` and `createBook`. `createBook` mutation accepts a single argument (`name`) and returns a newly created `Author` object. `createBook` mutation accepts two arguments (`name` and `authorId`) and returns a newly created `Book` object.

Structuring a mutation

Like queries, mutations match the structure of your schema's type definitions. The following mutation creates a new `Author` and requests certain fields of the created object as a return value:

```
mutation CreateBook {
  createAuthor(name: "Leo Tolstoy") {
    id
    name
  }
}
```

As with queries, our server would respond to this mutation with a result that matches the mutation's structure, like so:

```
{
   "data": {
     "createAuthor": {
       "id": 1,
       "name": "Leo Tolstoy",
     }
   }
 }
```

Resolvers

GraphQL Server needs to know how to populate data for every field in your schema so that it can respond to requests for that data. To accomplish this, it uses resolvers.

A resolver is a function that's responsible for populating the data for a single field in your schema. It can populate that data in any way you define, such as by fetching data from a back-end database or a third-party API.

Defining Resolvers

Let's say our server defines the schema described above. We want to define resolvers for `Query`, `Mutation`, `Author`, `Book` and their fields.

GraphQL for Delphi library can map GraphQL schema definition onto Delphi class structure. Those resolvers definitions look like this:

```
type
  TBook = class;

  TAuthor = class
  private
    FId: Integer;
    FName: string;
    function GetBooks: TArray<TBook>;
  public
    constructor Create(AName: string);
    property Books: TArray<TBook> read GetBooks;

    property Id: Integer read FId;
    property Name: string read FName;
  end;

  TBook = class
  private
    FId: Integer;
    FName: string;
    FAuthorId: Integer;
    function GetAuthor: TAuthor;
  public
    constructor Create(AName: string; AAuthorID: Integer);
    property Id: Integer read FId;
    property Name: string read FName;
    property Author: TAuthor read GetAuthor;
  end;

  TMutation = class
  public
    function CreateAuthor(Name: string): TAuthor;
    function CreateBook(Name: string; AuthorID: Integer): TBook;
  end;

  TQuery = class
  public
    function Books: TArray<TBook>;
    function Authors: TArray<TAuthor>;
    function Book(id: Integer): TBook;
    function Author(id: Integer): TAuthor;
  end;
```

Release Notes

Version 1.5 (Sep-2025)

- **New: Delphi 13 support.**

Version 1.4 (Jan-2024)

- **New: Delphi 12 support.**
- **New: Added support for TMS Sparkle, allowing to create GraphQL servers using Sparkle instead of Web Broker.**
- **New: Added support for type extensions in GraphQL parser.**
- **Improved**: New TFieldResolverArgs.Arguments property provides information about the field arguments.
- **Improved**: Responses are being cached for increased performance.
- **Improved**: Improved performance for retrieval of introspection information from the server.
- **Fixed**: Empty lists were being rejected for non-null lists.

Version 1.3 (Apr-2022)

- **New**: TSchemaDocument.OnGlobalCreate event allows for registration of custom types in schema document, especially scalar types.
- **Improved**: Bookshelf demo updated showing how to create custom scalar types in GraphQL schema.

Version 1.2 (Mar-2022)

- **Fixed**: Validation issue when parameter types were input lists.

Version 1.1 (Feb-2022)

- **New: TGraphQLExecuter.Current property**.

Version 1.0 (Feb-2022)

- First release.

Breaking Changes

List of changes in each version that breaks backward compatibility from a previous version.

No breaking changes so far.

Copyright Information

Main Copyright

Download

GraphQL for Delphi is distributed and supported by TMS Software. For detailed information about support, downloads, licensing, editions, and, in summary to get answers to any questions you might have regarding GraphQL for Delphi, you can refer to the link below:

GraphQL for Delphi official page at TMS Software

GraphQL for Delphi installer download links

You can use GraphQL for Delphi starting in Delphi 10 Seattle and up to the latest Delphi versions. It has two different distributions: Free Edition and the Registered Edition.

You can download the installers for Free Edition right away from GraphQL for Delphi download page. Alternatively you can click the following download links directly:

- Delphi 13 Florence
- Delphi 12 Athens
- Delphi 11 Alexandria
- Delphi 10.4 Sydney
- Delphi 10.3 Rio
- Delphi 10.2 Tokyo
- Delphi 10.1 Berlin
- Delphi 10 Seattle

Just download and run the installer for the Delphi version you use.

GraphQL for Delphi is also available in GetIt. Go to Delphi menu option `Tools | GetIt Package Manager...` and search for `graphql`. Select GraphQL for Delphi from the list of available filtered packages and click "Install".

Installers for Registered Edition are made available directly from customer area of TMS Software web site. You can purchase a Registered Edition license directly from the GraphQL for Delphi page at TMS Software. The page also provides a comparison table displaying the differences between the Free Edition and Registered Edition.