TutorialEdge.net

- The Go Path
- Courses
- Tutorials
- Tips

🚀 Get 25% off access to all my premium courses - use discount code FUNCMAIN at checkout - view the pricing page now!

Go GraphQL Beginners Tutorial

#graphql

Elliot Forbes ⏰ 10 Minutes 📅 Dec 27, 2018

Welcome fellow Gophers! In this tutorial, we are going to be looking at how we can interact with a GraphQL server within our Go-based programs. By the end of this tutorial, we should hopefully know how to do the following:

- The basics of GraphQL
- Build a Simple GraphQL Server in Go
- Perform basic queries against GraphQL

We'll be focused on the data-retrieval side of GraphQL in this tutorial and we'll back it with an in-memory data source. This should give us a good base to build up on top of in subsequent tutorials.

The Basics of GraphQL

Ok, so before we dive in, we should really cover the basics of GraphQL. How does using it benefit us as developers?

Well, consider working with systems that handle hundreds of thousands, if not millions of requests per day. Traditionally, we would hit an API that fronts our database and it would return a massive JSON response that contains a lot of redundant information that we might not necessarily need.

If we are working with applications at a massive scale, sending redundant data can be costly and choke our network bandwidth due to payload size.

GraphQL essentially allows us to cut down the noise and describe the data that we wish to retrieve from our APIs so that we are retrieving *only* what we require for our current task/view/whatever.

This is just one example of the many benefits the technology provides us. Hopefully, in the coming tutorial series, we'll see a few more of these benefits up front.

Query Language for our API, NOT our Database

One important thing to note is that GraphQL is not a query language like our traditional SQL. It is an abstraction that sits in-front of our APIs and is **not** tied to any specific database or storage engine.

This is actually really cool. We can stand up a GraphQL server that interacts with existing services and then build around this new GraphQL server instead of having to worry about modifying existing REST APIs.

Comparing REST to GraphQL

Let's look at how the RESTful approach differs from the GraphQL approach. Now, imagine we were building a service that returns all of the tutorials on this site, if we wanted a particular tutorial's information, we would generally create an API endpoint that allowed us to retrieve particular tutorials based on an ID:

```
## A dummy endpoint that takes in an ID path parameter
'http://api.tutorialedge.net/tutorial/:id'
```

This would then return a response, if given a valid `ID`, that would look something like this:

```
{
  "title": "Go GraphQL Tutorial",
  "Author": "Elliot Forbes",
  "slug": "/golang/go-graphql-beginners-tutorial/",
  "views": 1,
  "key": "value"
}
```

Now, say we wanted to create a widget that listed the top 5 posts written by said author. We could hit the `/author/:id` endpoint to retrieve all of the posts written by that author and then make subsequent calls to retrieve each of the top 5 posts. Or, we could craft an entirely new endpoint which returns this data for us.

Neither solution sounds particularly appealing as they create an unneeded amount of request or return too much data, and this highlights where the RESTful approach starts to present a few cracks.

This is where GraphQL comes into play. With GraphQL, we can define the exact structure of the data we want returned in the Query. So if we wanted the above information, we could create a query that looked something like so:

```
{
    tutorial(id: 1) {
        id
        title
        author {
            name
            tutorials
        }
        comments {
            body
        }
    }
}
```

This would subsequently return our tutorial, the author of said tutorial and an array of tutorial IDs representing the tutorials written by this author without having to send an additional *x* more REST requests to get the information! How good is that?

Basic Setup

Ok, so now that we understand a little bit more about GraphQL and how it's useful, let's see it in practice.

We are going to be creating a simple GraphQL server in Go, using the graphql-go/graphql implementation.

Setting up a Simple Server

Let's start by initializing our project using `go mod init`:

```
$ go mod init github.com/elliotforbes/go-graphql-tutorial
```

Next, let's create a new file called `main.go`. We'll start off simple and create a really simple GraphQL server that features a really simple resolver:

```
// credit - go-graphql hello world example
package main

import (
    "encoding/json"
    "fmt"
    "log"

    "github.com/graphql-go/graphql"
)

func main() {
    // Schema
    fields := graphql.Fields{
        "hello": &graphql.Field{
            Type: graphql.String,
            Resolve: func(p graphql.ResolveParams) (interface{}, error) {
                return "world", nil
            },
        },
    }
    rootQuery := graphql.ObjectConfig{Name: "RootQuery", Fields: fields}
    schemaConfig := graphql.SchemaConfig{Query: graphql.NewObject(rootQuery)}
    schema, err := graphql.NewSchema(schemaConfig)
    if err != nil {
        log.Fatalf("failed to create new schema, error: %v", err)
    }

    // Query
    query := `
        {
            hello
        }
    `
    params := graphql.Params{Schema: schema, RequestString: query}
    r := graphql.Do(params)
    if len(r.Errors) > 0 {
        log.Fatalf("failed to execute graphql operation, errors: %+v", r.Errors)
    }
    rJSON, _ := json.Marshal(r)
    fmt.Printf("%s \n", rJSON) // {"data":{"hello":"world"}}
}
```

Now, if we try and run this, let's see what happens:

```
$ go run ./...
{"data":{"hello":"world"}}
```

So, if everything worked, then we've been able to set up a really simple GraphQL server and make a really simple query to this server.

GraphQL Schema

Let's break down what was going on in the above code so that we can expand it further. On `lines 14-21` we define our `Schema`. When we make queries against our GraphQL API, we essentially define what fields on objects we want returned to us, so we have to define these fields within our Schema.

On `line 17`, we define a resolver function that is triggered whenever this particular `field` is requested. Right now, we are just returning the string `"world"`, but we'll be implementing the ability to query the database from here.

Querying

Let's have a look at the second part of our `main.go` file. On `line 30` we start to define a `query` that requests the field `hello`.

We then create a `params` struct which contains a reference to our defined `Schema` as well as our `RequestString` request.

Finally, on `line 36` we execute the request and the results of the request are populated into `r`. We then do some error handling and then Marshal the response into JSON and print it out to our console.

A More Complex Example

Now that we have a really simple GraphQL server up and running and we are able to query against it, let's take it a step further and build a more complex example.

We'll be creating a GraphQL server that returns a series of in-memory tutorials as well as their Author, and any comments made on those particular tutorials.

Let's define some `struct`'s that will represent a `Tutorial`, an `Author`, and a `Comment`:

```
type Tutorial struct {
    Title    string
    Author   Author
    Comments []Comment
}

type Author struct {
    Name      string
    Tutorials []int
}

type Comment struct {
    Body string
}
```

We can then create a really simple `populate()` function which will return an array of type `Tutorial`:

```
func populate() []Tutorial {
    author := &Author{Name: "Elliot Forbes", Tutorials: []int{1}}
    tutorial := Tutorial{
        ID:     1,
        Title:  "Go GraphQL Tutorial",
        Author: *author,
        Comments: []Comment{
            Comment{Body: "First Comment"},
        },
    }

    var tutorials []Tutorial
    tutorials = append(tutorials, tutorial)

    return tutorials
}
```

This will give us a simple list of tutorials that we can then resolve to later on.

Creating New Object Types

We'll start off by creating a new object in GraphQL using `graphql.NewObject()`. We'll define 3 different Types using GraphQL's strict typing, these will match up with the 3 `structs` we've already defined.

Our `Comment` struct is arguably our simplest, it contains just a string `Body`, so we can represent this as a `commentType` fairly easily like so:

```
var commentType = graphql.NewObject(
    graphql.ObjectConfig{
        Name: "Comment",
        // we define the name and the fields of our
        // object. In this case, we have one solitary
        // field that is of type string
        Fields: graphql.Fields{
            "body": &graphql.Field{
                Type: graphql.String,
            },
        },
    },
)
```

Next, we'll tackle the `Author` struct and define that as a new `graphql.NewObject()`. This will be slightly more complex as it features both a `String` field, as well as a list of `Int` values that represent the ID's of the tutorials that they have written.

```
var authorType = graphql.NewObject(
    graphql.ObjectConfig{
        Name: "Author",
        Fields: graphql.Fields{
            "Name": &graphql.Field{
                Type: graphql.String,
            },
            "Tutorials": &graphql.Field{
                // we'll use NewList to deal with an array
                // of int values
                Type: graphql.NewList(graphql.Int),
            },
        },
    },
)
```

And finally, let's define our `tutorialType` which will encapsulate both an `author`, and a array of `comment`'s as well as an `ID` and a `title`:

```
var tutorialType = graphql.NewObject(
    graphql.ObjectConfig{
        Name: "Tutorial",
        Fields: graphql.Fields{
            "id": &graphql.Field{
                Type: graphql.Int,
            },
            "title": &graphql.Field{
                Type: graphql.String,
            },
            "author": &graphql.Field{
                // here, we specify type as authorType
                // which we've already defined.
                // This is how we handle nested objects
                Type: authorType,
            },
            "comments": &graphql.Field{
                Type: graphql.NewList(commentType),
            },
        },
    },
)
```

Updating our Schema

Now that we've defined our `Type` system, let's set about updating our Schema to reflect these new types. We'll define 2 distinct `Field`'s, the first will be our `tutorial` field which will allow us to retrieve individual `Tutorials` based on an ID passed in to the query. The second will be a `list` field which will allow us to retrieve the full array of `Tutorials` that we have defined in memory.

```
    // Schema
    fields := graphql.Fields{
        "tutorial": &graphql.Field{
            Type:        tutorialType,
            // it's good form to add a description
            // to each field.
            Description: "Get Tutorial By ID",
            // We can define arguments that allow us to
            // pick specific tutorials. In this case
            // we want to be able to specify the ID of the
            // tutorial we want to retrieve
            Args: graphql.FieldConfigArgument{
                "id": &graphql.ArgumentConfig{
                    Type: graphql.Int,
                },
            },
            Resolve: func(p graphql.ResolveParams) (interface{}, error) {
                // take in the ID argument
                id, ok := p.Args["id"].(int)
                if ok {
                    // Parse our tutorial array for the matching id
                    for _, tutorial := range tutorials {
                        if int(tutorial.ID) == id {
                            // return our tutorial
                            return tutorial, nil
                        }
                    }
                }
                return nil, nil
            },
        },
        // this is our `list` endpoint which will return all
        // tutorials available
        "list": &graphql.Field{
            Type:        graphql.NewList(tutorialType),
            Description: "Get Tutorial List",
            Resolve: func(params graphql.ResolveParams) (interface{}, error) {
                return tutorials, nil
            },
        },
    }
```

So we’ve created our Types and updated our GraphQL schema, we aren’t doing too bad!

Testing it Works

Let’s try playing with our new GraphQL server and play around with the queries that we are submitting. Let’s try out our `list` schema by changing the `query` we’ve got in our `main()` function:

```
// Query
query := `
    {
        list {
            id
            title
            comments {
                body
            }
            author {
                Name
                Tutorials
            }
        }
    }
`
```

Let’s break this down. So within our query we have a special `root` object. Within this we then say that we want the `list` field on that object. On the list returned by `list`, we want to see the `id`, `title`, `comments` and the `author`.

When we go to run this, we should then see the following output:

```
$ go run ./...
{"data":{"list":[{"author":{"Name":"Elliot Forbes","Tutorials":[1]},"comments":[{"body":"First Comment"}],"id":1,"title":"Go GraphQL Tutorial"}]}
```

As we can see, our query has returned all of our tutorials, in a JSON form that looks very much like the structure of our initial Query.

Let’s now try a query against our `tutorial` schema:

```
query := `
    {
        tutorial(id:1) {
            title
            author {
                Name
                Tutorials
            }
        }
    }
`
```

And again, when we run this, we should see that it has successfully retrieved the solitary tutorial in memory with the `ID=1`:

```
$ go run ./...
{"data":{"tutorial":{"author":{"Name":"Elliot Forbes","Tutorials":[1]},"title":"Go GraphQL Tutorial"}}}
```

Perfect, it looks like we’ve gotten both our `list` and our `tutorial` schema working as expected.

> **Challenge -** Try updating the list of tutorials within our `populate()` function so that it returns more tutorials. Once we’ve done this, play around with the queries and try to become more familiar with them.

Conclusion

> **Note -** The full source code for this tutorial can be found here: main.go

That's all we'll be covering in this initial tutorial. We've managed to successfully set up a simple GraphQL server that is backed by an in-memory data-store.

In the next tutorial, we'll be looking at GraphQL mutations and changing our data-source to use a SQL database. The next part of this tutorial series can be found here: Go GraphQL Beginners Tutorial - Part 2

> **Note -** If you want to keep track of when new Go articles are posted to the site, then please feel free to follow me on twitter for all the latest news: @Elliot_F.

Previous Article

Go Graphql Beginners Tutorial - Part 2

Next Article

An Introduction to Go Closures - Tutorial

💬 Discussion

Always be kind when commenting and adhere to our Code of Conduct

Become a Member or Log In

🌲 Carbon Negative Learning 🌲

A percentage of your monthly subscription goes towards planting trees and helping to regenerate forests around the world. View our pricing options: Pricing

Trees Planted:

🌲 13738

Carbon Offset:

🌍 121.86 Tonnes

TutorialEdge

TutorialEdge is a rapidly growing site focused on delivering high quality, in-depth courses on Go.

New videos are added at the end of every week and a roughly 10% of the site's revenue goes towards tackling climate change through tree planting and carbon capture initiatives.

About

- Blog
- Privacy Policy
- Support Policy
- Code of Conduct
- Pricing

Top Courses

- Building a Production-Ready REST API in Go
- RabbitMQ Crash Course for Go
- The Go Testing Bible

TutorialEdge

Sign up for a free account and attempt the growing selection of challenges up on the site!

Sign In 🚀

Legal

Support Policy Code of Conduct Pricing Privacy Policy

© TutorialEdge

My Scottish Adventure Photography Blog Carbon Negative 🌲