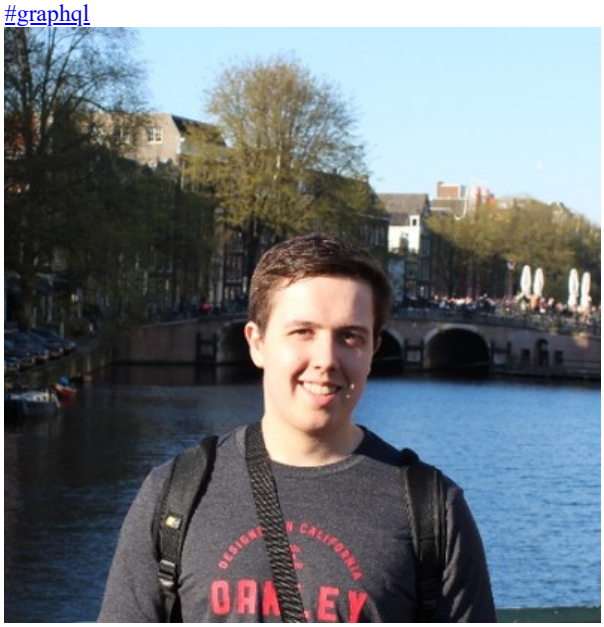
TutorialEdge.net

Go Graphql Beginners Tutorial - Part 2

#graphql



Elliot Forbes ⏰ 10 Minutes 🗓️ Dec 28, 2018

- Mutations
  - A New Tutorial
- Swapping out Data Sources
  - A Simple MySQL Database
  - Retrieving a Single Tutorial
  - Key Points
  - Using an ORM
- Conclusion

Note - This tutorial is part of a 2-part mini-series on GraphQL, the first part of this tutorial can be found here: Go GraphQL Beginners Tutorial - Part 1

Welcome fellow Gophers! In this tutorial, we are going to be expanding upon the work we did in our previous GraphQL Go tutorial and looking at mutations and implementing proper data-sources behind our GraphQL API.

In the previous tutorial, we looked at some of the major benefits of GraphQL and how it could greatly improve the way we retrieve data for particular components within our applications.

We looked at why building an orchestration API for particular views was potentially wasteful and how this new technology allows you to reduce the noise of the data retrieved by our web components.

> **Reading Material -** If you are unconvinced about the benefits of GraphQL, I would recommend checking out this really interesting post from Paypal Engineering which highlights their successes with the technology - GraphQL - A Success Story for Paypal Checkout

Mutations

Let's dive in, the first thing we are going to be looking at is mutations. We'll do this before we cover data-sources as it is far easier to learn these concepts without having to also craft SQL queries on top of that.

Mutations in GraphQL allow us to not only fetch data from our GraphQL API, but to also update it. This ability to modify our data makes GraphQL a far more complete data platform that could entirely replace REST as opposed to just complement it.

These *mutations* follow a very similar structure to the queries that we crafted in part 1 of this mini-series. Not only that, but defining them within our code also follows an incredibly similar structure.

Each of our *mutations* will be mapped with a distinct *resolver* function which will perform the updates to the data that we require. For instance, if we wanted, we could create a *mutation* that allowed me to update a particular tutorial, this would take in all of the information within the mutation query and it would resolve to a single *resolver* function which would go away and update this tutorial in the database with the new information.

A New Tutorial

Let's start off by trying to create a really simple mutation that will allow us to add new tutorials to our existing list of tutorials.

We'll create a new GraphQL Object, very much like we did with our `fields` and within this new object we'll define our `create` field. This `create` field will simply populate a new Tutorial based on the arguments passed in to it, and then append this new Tutorial to our global array of tutorials.

> **Some Slight Tweaks -** We'll need to move `tutorials` to a global variable to make this easy for us. This will later be replaced by a database so don't worry too much about setting/modifying global state.

```
var mutationType = graphql.NewObject(graphql.ObjectConfig{
    Name: "Mutation",
    Fields: graphql.Fields{
        "create": &graphql.Field{
            Type:        tutorialType,
            Description: "Create a new Tutorial",
            Args: graphql.FieldConfigArgument{
                "title": &graphql.ArgumentConfig{
                    Type: graphql.NewNonNull(graphql.String),
                },
            },
            Resolve: func(params graphql.ResolveParams) (interface{}, error) {
                tutorial := Tutorial{
                    Title: params.Args["title"].(string),
                }
                tutorials = append(tutorials, tutorial)
                return tutorial, nil
            },
        },
    },
})
```

We'll then have to update our `schemaConfig` within our `main()` function to reference our new `mutationType`:

```
schemaConfig := graphql.SchemaConfig{
    Query:    graphql.NewObject(rootQuery),
    Mutation: mutationType,
}
```

Now, we have created a simple example of a *mutation*, and we've updated our `schemaConfig`, we can then try using this new *mutation*.

Let's update the `query` that we have within our `main()` function to use this new *mutation*:

```
// Query
query := `
    mutation {
        create(title: "Hello World") {
            title
        }
    }
`
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)
```

And, to check that everything has worked as expected, we'll quickly create and execute a second `query` below this one to fetch a `list` of all the tutorials we have in-memory after our *mutation* has been called.

```
// Query
query = `
    {
        list {
            id
            title
        }
    }
`
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)
```

Full Source Code - The full source code for this section of the tutorial can be found here: simple-mutation.go

When we attempt to run this, we should see that our *mutation* is successfully called and that our returned list of Tutorials now includes our newly defined tutorial:

```
$ go run ./...
{"data":{"create":{"title":"Hello World"}}}
{"data":{"list":[{"id":1,"title":"Go GraphQL Tutorial"},{"id":2,"title":"Go GraphQL Tutorial - Part 2"},{"id":0,"title":"Hello World"}]}}
```

Awesome, we have now successfully created a mutation that allows us to insert new tutorials into our existing list of tutorials.

If we wanted to take this further, we could add more mutations that update existing tutorials or even delete them from our list. We would simply create a new `field` on our mutation object that features it's own `resolve` function for each of these actions, and then within these `resolve` functions we would implement the logic for updating/deleting particular tutorials.

Swapping out Data Sources

Now that we've covered mutations, it is probably a good time to cover how we would swap out our in-memory data store for something like MySQL or MongoDB.

An awesome feature of GraphQL is that it isn't constrained to any particular set of database technologies. We can create a GraphQL API that interacts with NoSQL as wel as SQL databases at will.

A Simple MySQL Database

For the purpose of this tutorial, we'll use a SQLite3 local SQL database to quickly demonstrate how you could swap in a more effective data-source.

We'll create the new database by calling:

```
$ sqlite3 tutorials.db
```

This will open up an interactive shell which we can use to manipulate and query our SQL database. We want to start off by creating a table `tutorials`:

```
sqlite> CREATE TABLE tutorials (id int, title string);
```

We then want to insert a couple of rows into our database so that we can verify our `List` query changes will work:

```
sqlite> INSERT INTO tutorials VALUES (1, "First Tutorial");
sqlite> INSERT INTO tutorials VALUES (2, "Second Tutorial");
sqlite> INSERT INTO tutorials VALUES (3, "third Tutorial");
```

Now that we have a simple database and a few rows, we can update code to connect out to this new database. We'll first need to add a new import to the top of our `main.go` file which will allow us to communicate with our `SQLite3` database:

```
package main

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

    "github.com/graphql-go/graphql"
    _ "github.com/mattn/go-sqlite3"
)
...
```

Now that we've added this new package, we can attempt to use it by updating our `list` Field that we defined within our `main` function.

There is nothing new here from a GraphQL perspective. We've simply swapped out the logic for returning a list of in-memory tutorials, to now query a database and populate a list of tutorials before returning them.

```
"list": &graphql.Field{
    Type:        graphql.NewList(tutorialType),
    Description: "Get Tutorial List",
    Resolve: func(params graphql.ResolveParams) (interface{}, error) {
        db, err := sql.Open("sqlite3", "./tutorials.db")
        if err != nil {
            log.Fatal(err)
        }
        defer db.Close()
        // perform a db.Query insert
        var tutorials []Tutorial
        results, err := db.Query("SELECT * FROM tutorials")
        if err != nil {
            fmt.Println(err)
        }
        for results.Next() {
            var tutorial Tutorial
            err = results.Scan(&tutorial.ID, &tutorial.Title)
            if err != nil {
                fmt.Println(err)
            }
            log.Println(tutorial)
            tutorials = append(tutorials, tutorial)
        }
        return tutorials, nil
    },
},
```

With these changes made, we can query our `list` to see if this worked. Update the `query` within your `main` function so that it queries `list` and retrieves the `id` and the `title` like so:

```
// Query
query := `
    {
        list {
            id
            title
        }
    }
`
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)
```

When we run this, we should see that 3 rows are returned from our sqlite3 database and we can see the JSON response from our GraphQL query. Again, if we wanted just the `id`'s of our Tutorials returned here, all we would have to do is modify our `query` to remove `title` and everything would work as expected.

```
$ go run ./...
2018/12/30 14:44:08 {1 First Tutorial { []} []}
2018/12/30 14:44:08 {2 Second Tutorial { []} []}
2018/12/30 14:44:08 {3 third Tutorial { []} []}
{"data":{"list":[{"id":1,"title":"First Tutorial"},{"id":2,"title":"Second Tutorial"},{"id":3,"title":"third Tutorial"}]}}
```

Retrieving a Single Tutorial

We've just about got the hang of using an external data-source for our GraphQL API, but let's now take a look at one more easy example. Let's have a look at updating our `tutorial` schema so that it now references our new sqlite3 data-source.

```
"tutorial": &graphql.Field{
    Type:        tutorialType,
    Description: "Get Tutorial By ID",
    Args: graphql.FieldConfigArgument{
        "id": &graphql.ArgumentConfig{
            Type: graphql.Int,
        },
    },
    Resolve: func(p graphql.ResolveParams) (interface{}, error) {
        id, ok := p.Args["id"].(int)
        if ok {
            db, err := sql.Open("sqlite3", "./tutorials.db")
            if err != nil {
                log.Fatal(err)
            }
            defer db.Close()
            var tutorial Tutorial
            err = db.QueryRow("SELECT ID, Title FROM tutorials where ID = ?", id).Scan(&tutorial.ID, &tutorial.Title)
            if err != nil {
                fmt.Println(err)
            }
            return tutorial, nil
        }
        return nil, nil
    },
},
```

All we are doing here is opening a new connection to our existing database and then querying that database for a single Tutorial with an `ID` equal to the value of the `ID` we pass in our `query`.

```
// Query
query := `
    {
        tutorial(id: 1) {
            id
            title
        }
    }
`
```

When we go to run this, we should see that our resolver function has successfully connected to our SQLite database and retrieved the existing tutorial:

```
$ go run ./...
{"data":{"tutorial":{"id":1,"title":"First Tutorial"}}}
```

Key Points

So, just to reiterate, we've moved away from parsing and returning an in-memory list of Tutorials and instead, we are connecting to a database, making a SQL query and populating our list of tutorials from this.

GraphQL handles everything after we've retrieved the results. If we wanted to return the author of each tutorial and the comments, we could create further tables within our SQLite database to store these. We could then simply perform additional SQL queries to our database to retrieve the author based on an `ID` as well as the comments.

Using an ORM

If we wanted to simplify our development work further, there is the option of using an ORM when communicating with our SQLite database. This would handle our SQL queries for us and handle any joins or additional queries that would have to be made if we had nested or joined elements such as our comments and our authors.

> **Note -** If you haven't encountered ORMs before or how you would use them in Go, I recommend checking out my other tutorial: Go ORM Tutorial

By using an ORM to handle the database retrieval / update model, we can somewhat simplify our code and continue to expand our API purely in Go. Again, I've created a reference implementation of this that you can view here: elliotforbes/go-graphql-tutorial

Conclusion

So, this concludes part 2 of my Go GraphQL mini-series. In this tutorial, we covered the basics of *mutations* and how you could swap out the data-sources backing our GraphQL APIs to use various different database technologies like SQL or NoSQL databases.

One of the things I hope you take away from this is the potential impact that GraphQL can have on your development velocity when it comes to implementing the applications that rely on these APIs.

By employing a technology such as GraphQL, we minimize the amount of time we spend figuring out how to fetch and parse data and subsequently we can focus this time on improving the UI/UX of our applications.

> **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

Working with Temporary Files and Directories in Go 1.11

Next Article

Go GraphQL Beginners Tutorial

Discussion

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

Become a Member or Log In

Ukeje Chukwuemeriwo Goodness @ 19 Jul, 2022 12:17

I'm curious as to why you didn't choose a schema first library. any reasons or it's just a preference

🌲 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

My Scottish Adventure Photography Blog Carbon Negative 🌲