Generating a GraphQL Server with gqlgen

Mathew Byrne
mathew.byrne@99designs.com
@mathewbyrne

Wi-Fi

99designs-guest ninety9d3signs

Workshop Material

https://github.com/99designs/gqlgen-workshop

Setup

For this workshop you will ideally have setup:

- **Go 1.8+**
- A **`$GOPATH`** setup in your environment — we will assume **`~/go`**
- **`$GOPATH/bin/`** in your **`$PATH`**
- Run **`gofmt`** on save

Setup — Dependencies

Create a folder for our project:

```
$ mkdir -p $GOPATH/src/github.com/[your-github-username]/gqlgen-workshop
$ cd $GOPATH/src/github.com/[your-github-username]/gqlgen-workshop
```

Install dependencies:

```
$ go get -v -u github.com/99designs/gqlgen/... \
    github.com/99designs/gqlgen-workshop/...
```

What is GraphQL?

- A query language for an API
- A runtime for fulfilling those queries
- Ask for your exact data requirements
- Schemas and type system

by Ashur

by RVST

How Does This Compare to REST?

- Multiple resources in a single query
- Reduce over-fetching
- Schema provides a complete description of an endpoint

gqlgen

- We looked at existing Go libraries doing GraphQL, the tradeoffs were not great:
  - `graphql/graphql-go` — has a DSL that was easy to break and hard to read
  - `graph-gophers/graphql-go` — very verbose, boilerplate, namespace collisions
  - general lack of features
- Goals of gqlgen:
  - Type safety without all the boilerplate
  - Good developer experience
  - Fast runtime

Adam Scarr
vektah

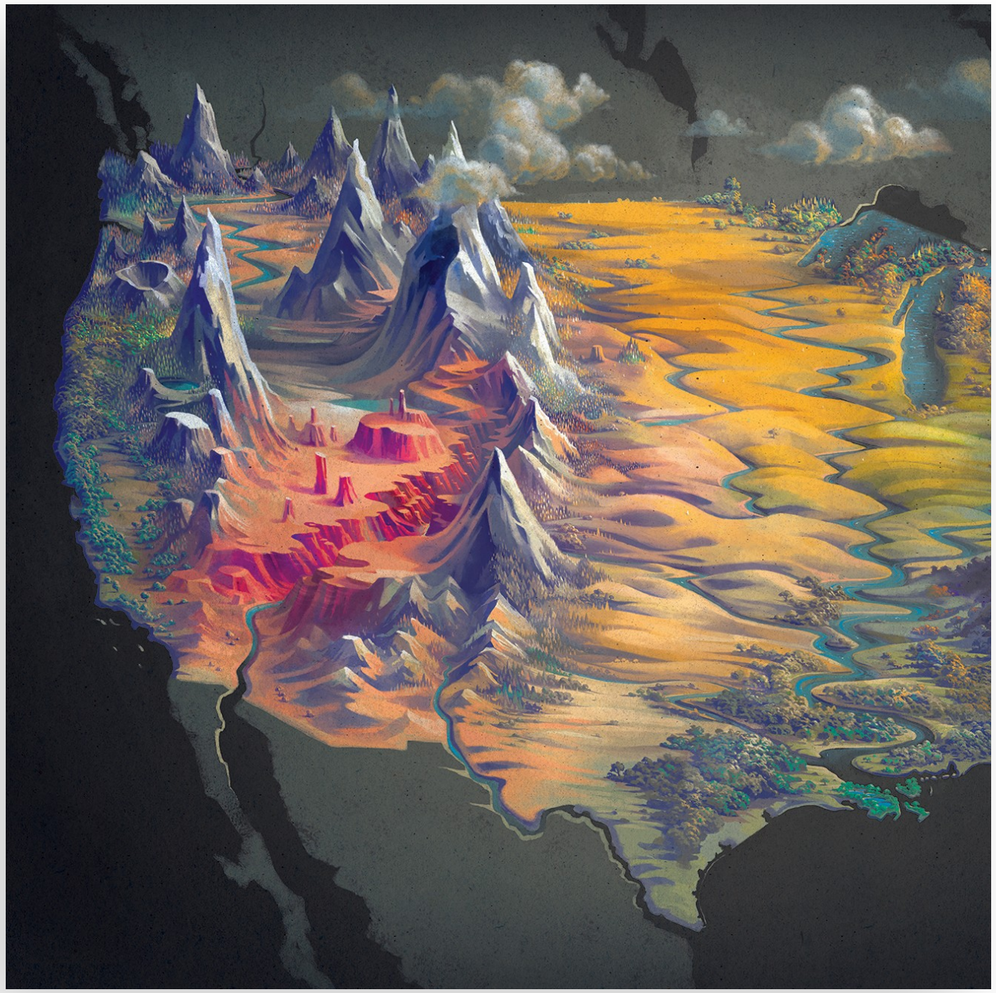
graph enthusiast at 99designs

What Are We Building?

- In this workshop we're going to build a graph that exposes:
  - a local mock database
  - a remove Movie API that we can search
  - a combination of both services into a single graph that allows users to like Movies
- This graph could be the starting point of a Movie liking application

Final Schema

```
type Query {
    user(id: ID!): User
    movies(search: String!): [Movie!]!
}

type User {
    id: ID!
    name: String!
    likes: [Movie!]!
}

type Movie {
    id: ID!
    title: String!
    year: String!
}

type Mutation {
    like(userId: ID!, movieId: ID!): User
}
```



Local Database

So first imagine we have a local app with a User entity:

```
type User struct {
    ID   int
    Name string
}
```

For this workshop we've provided this in the db package which also exposes these methods:

```
func GetUser(id int) *User
```

GraphQL Schema

To generate our GraphQL server, we first need schema.

Create a file named **`schema.graphql`**

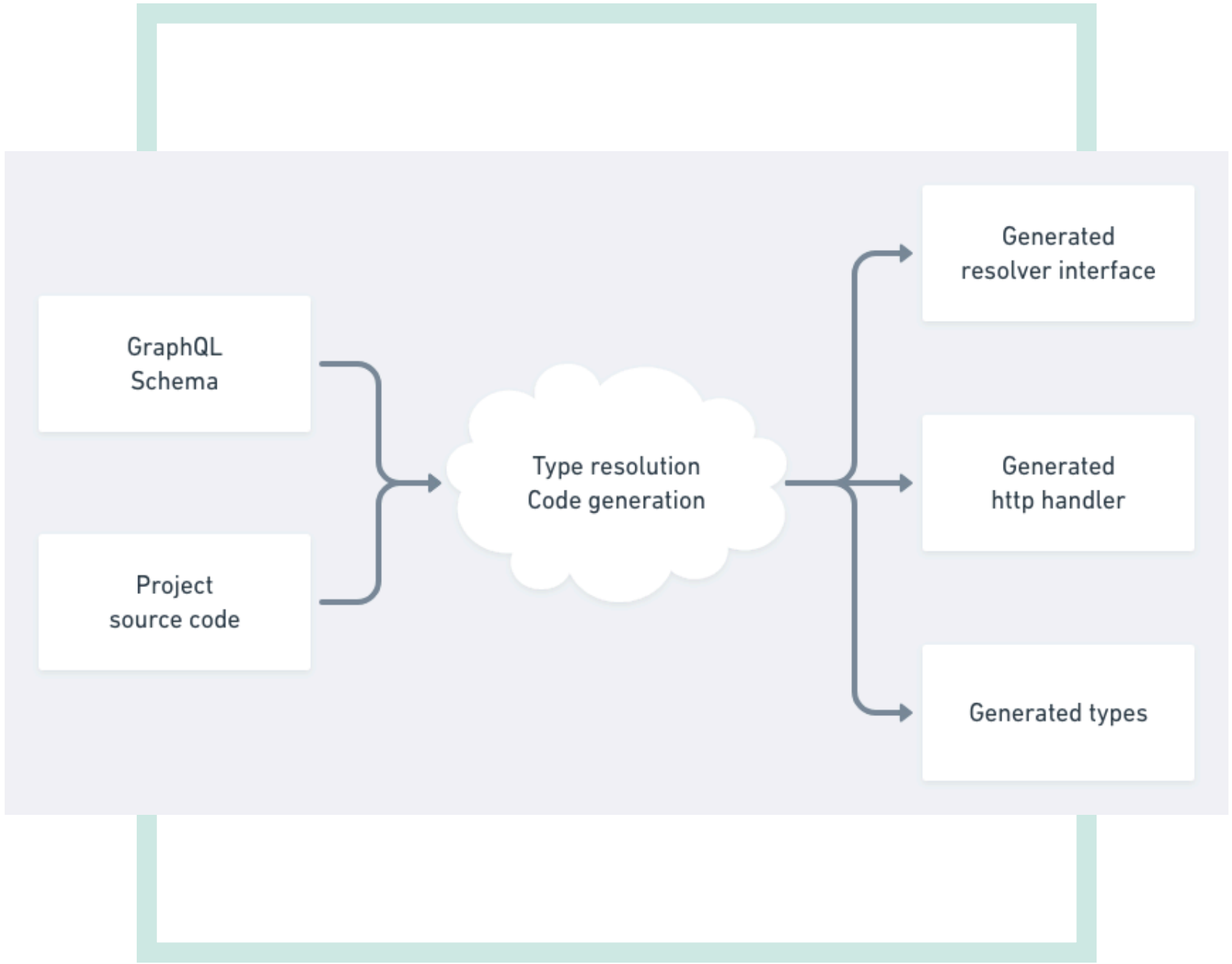
```
type Query {
    user(id: ID!): User
}

type User {
    id: ID!
    name: String!
}
```

Initialise gqlgen

```
$ gqlgen init
```

After this your project directory should look like:

```
$ ls
generated.go   gqlgen.yml     models_gen.go  resolver.go    schema.graphql server
```



Type Mapping

- `gqlgen` will handle mapping GraphQL types to Go
- By default it will generate new types for us
- Can be configured to map to existing types

Type Mapping Setup

gqlgen has generated a model for us in `models_gen.go` but we have a User model already, so let's map to that.

Edit `gqlgen.yml`

```
models:
  User:
    model: github.com/99designs/gqlgen-workshop/db.User
```

And generate. We're also going to remove resolver so that it's updated to point to our new package.

```
$ rm resolver.go
$ gqlgen
```

Implement User Resolver

Now we can connect to our backend through our generated resolver. Add the following implementation to `resolver.go`

```
func (r *queryResolver) User(ctx context.Context, id string) (*db.User, error) {
    user := db.GetUser(id)
    if user == nil {
        return nil, errors.New("User not found")
    }
    return user, nil
}
```

Boot the Server!

```
$ go run server/server.go
```

Open http://localhost:8080/ and try a query:

```
query {
    user(id:"1") {
        name
    }
}
```

External Services

The provided package `github.com/99designs/gqlgen-workshop/omdb` exposes an API client we can use to query a public movie database:

```
type Movie struct {
    ID    string
    Title string
    Year  string
}

func Search(term string) ([]Movie, error)
```

We're now going to expose this API through our graph and integrate it with our User type.

Movie — Schema

First we should update our schema. Add a new Type to `schema.graphql`

```
type Movie {
    id: ID!
    title: String!
    year: String!
}
```

Then add a movie edge to the root query:

```
type Query {
    # ...
    movies(search: String!): [Movie!]!
}
```

Movie — Type Mapping

Since we have a third party type, instead of having `gqlgen` generate a model for us, we can instead configure the type mapping in `gqlgen.yml`:

```
models:
  Movie:
    model: github.com/99designs/gqlgen-workshop/omdb.Movie
```

And regenerate:

```
$ gqlgen
```

Movie — Resolver Implementation

Now we need to implement the new resolver that has been generated for us:

```
func (r *queryResolver) Movies(ctx context.Context, search string) ([]omdb.Movie, error) {
    return omdb.Search(search)
}
```

Running the server now, we can query for Movies!

```
query {
    movies(search:"Star Wars") {
        id
        title
    }
}
```

Likes

Now let's say we want to enable users to like movies — we need a mutation for the like action, and we need to expose likes for a user. First we'll add our mutation; update `schema.graphql`:

```
type Mutation {
    like(userId: ID!, movieId: ID!): User
}
```

And regenerate with `gqlgen`

Likes — Mutation Resolver

`gqlgen` has now generated a `MutationResolver` interface for us. We can implement this like so in `resolver.go`

```
type mutationResolver struct{ *Resolver }

func (r *Resolver) Mutation() MutationResolver {
    return &mutationResolver{r}
}

func (r *mutationResolver) Like(ctx context.Context, userId string, movieId string)
(*db.User, error) {
    user := db.GetUser(userId)
    if user == nil {
        return nil, errors.New("User not found")
    }
    user.Like(movieId)
    return user, nil
}
```

Likes — Retrieving For User

If we can like a movie for a user, we should then be able to retrieve all movies they have previously liked. Let's add this ability to our `schema.graphql`

```
type User {
    // ...
    likes: [Movie!]!
}
```

And regenerate with `gqlgen`

Likes — Retrieving Resolver

By default, `gqlgen` will generate us resolvers for any fields in our schema that it does not know about.

```
type UserResolver interface {
    Likes(ctx context.Context, obj *db.User) ([]omdb.Movie, error)
}
```

Implement this interface in `resolver.go`

```
type userResolver struct{ *Resolver }

func (r *Resolver) User() UserResolver {
    return &userResolver{r}
}

func (r *userResolver) Likes(ctx context.Context, u *db.User) ([]omdb.Movie, error) {
    return omdb.GetAll(u.Likes)
}
```

Give it a Shot!

With everything in place, we can now test it out. Start the server again and run this query:

```
mutation {
    like(userId:"1", movieId:"tt0076759") {
        name
        likes {
            title
        }
    }
}
```

You should now be able to search for a movie title and add it to the list of liked movies for a given user.

Advanced Topics

- **Directives** — a way of “tagging” parts of your schema to declaratively add functionality
- **Subscriptions** — subscribing to data changes on the server via web sockets
- **Testing** —

by Sergey Gudz

We're Hiring!

http://99designs.com/open-source


99designs