GRAPHQL FOR RAILS DEVELOPERS

RYAN BIGG

GraphQL for Rails Developers

Build a GraphQL API for Your Rails App

Ryan Bigg

Table of Contents

1. Getting started with GraphQL 1

1.1. Creating a new Rails application 1
1.2. Adding RSpec 2
1.3. Adding the GraphQL gem 3
1.4. Using our GraphQL schema 6
1.5. How the schema fits together 8

Chapter 1. Getting started with GraphQL

In this short first chapter, we're now going to create a new Rails app and add the `graphql` gem to that app. We'll then run the `graphql` gem's in-built generator and setup an Active Record model so that we have something to use in our GraphQL API.

1.1. Creating a new Rails application

The application that we'll be working on during this book is an application called Repo Hero. It's going to be a review aggregator for Git repositories. Users will be able to add their favorite repositories, categorize them and to leave reviews for those repositories. All of this will be enabled through our GraphQL API by the end of this book.

To get us started, we'll make sure we have the right version of Rails installed:

```
gem install rails -v 7.1.0
```

And then we can generate this new application:

```
rails _7.1.0_ new --api --minimal repo_hero
```

The `--api` and `--minimal` options passed to Rails here will make the application as small as it can be, only including the bits we want, and nothing that we don't want.

1.2. Adding RSpec

To test this application and all it does, we're going to use RSpec[1]. We're setting that up here so that when we run generators later on they will be generated with RSpec files, rather than Test::Unit files.

To setup RSpec, we'll run this command:

```
bundle add rspec-rails --group "development, test"
```

Once this command as finished, we'll now run the RSpec installer:

```
rails g rspec:install
```

This will add the base RSpec files to our application:

```
create  .rspec
create  spec
create  spec/spec_helper.rb
create  spec/rails_helper.rb
```

With this gem now setup, we'll work on adding the `graphql` gem to our application.

1.3. Adding the GraphQL gem

The `graphql` gem is *the* gem for building GraphQL APIs within Ruby applications. It's used by companies such as Shopify and GitHub. And now Repo Hero!

We can install the `graphql` gem by using the wonderful `bundle add` command again:

```
bundle add graphql
```

Once that gem is installed, then we can run a Rails generator that the gem provides. This generator will set up the GraphQL structure that our application needs:

```
rails g graphql:install
```

This generator generates quite a few different files:

```
      create  app/graphql/types
      create  app/graphql/types/.keep
      create  app/graphql/repo_hero_schema.rb
      create  app/graphql/types/base_object.rb
      create  app/graphql/types/base_argument.rb
      create  app/graphql/types/base_field.rb
      create  app/graphql/types/base_enum.rb
      create  app/graphql/types/base_input_object.rb
      create  app/graphql/types/base_interface.rb
      create  app/graphql/types/base_scalar.rb
      create  app/graphql/types/base_union.rb
      create  app/graphql/types/query_type.rb
add_root_type  query
      create  app/graphql/mutations
      create  app/graphql/mutations/.keep
      create  app/graphql/mutations/base_mutation.rb
      create  app/graphql/types/mutation_type.rb
```

```
add_root_type  mutation
      create  app/controllers/graphql_controller.rb
       route  post "/graphql", to: "graphql#execute"
Skipped graphiql, as this rails project is API only
  You may wish to use GraphiQL.app for development:
https://github.com/skevy/graphiql-app
      create  app/graphql/types/node_type.rb
      insert  app/graphql/types/query_type.rb
      create  app/graphql/types/base_connection.rb
      create  app/graphql/types/base_edge.rb
      insert  app/graphql/types/base_object.rb
      insert  app/graphql/types/base_object.rb
      insert  app/graphql/types/base_union.rb
      insert  app/graphql/types/base_union.rb
      insert  app/graphql/types/base_interface.rb
      insert  app/graphql/types/base_interface.rb
      insert  app/graphql/repo_hero_schema.rb
```

We'll see what most of these files do in due time. For now, we're setting them up so that we will have access to them later.

The main thing that we should care about here is that there's now a route that has been added to `config/routes.rb`:

Listing 1. config/routes.rb

```
post "/graphql", to: "graphql#execute"
```

This route sets up our GraphQL entrypoint. Whenever we perform a GraphQL operation, we'll be making a `POST` request to `/graphql`. GraphQL uses a `POST` request so that it can support long request bodies, which is something that a `GET` request does not do.[2]. This is different to how you might understand request routing within a Rails application where *read* operations are typically `GET` requests, and *write* operations are typically `POST`, `PUT`, `PATCH` or `DELETE`. This is a quirk of working with GraphQL, and a substantial departure for what those familiar with REST APIs might

expect - but ultimately it's neither good nor bad. GraphQL has ways of differentiating between read and write operations without using HTTP methods, and we'll see how that works later on.

This entrypoint points at a controller called `GraphqlController`, and an action within that controller called `execute`. Let's take a look at that action now:

Listing 2. app/controllers/graphql_controller.rb

```
  def execute
    variables = prepare_variables(params[:variables])
    query = params[:query]
    operation_name = params[:operationName]
    context = {
      # Query context goes here, for example:
      # current_user: current_user,
    }
    result = RepoHeroSchema.execute(query, variables: variables, context:
context, operation_name: operation_name)
    render json: result
  rescue StandardError => e
    raise e unless Rails.env.development?
    handle_error_in_development(e)
  end
```

This action takes in some parameters from the request, and passes them to a class called `RepoHeroSchema` and its `execute` method. This method serves as an entrypoint into the world of GraphQL for our application. The call to this schema doesn't need to happen from within a controller and to demonstrate that we'll now break from this walkthrough into a demonstration of how we can execute GraphQL operations using the `RepoHeroSchema`, outside the context of a controller.

1.4. Using our GraphQL schema

To use our GraphQL schema, we're going to write a small script. Just to really nail down that we can use this thing outside the context of a controller. We can create a small script called `graphql-test.rb` at the root of our Rails application.

Listing 3. graphql-test.rb

```
operation = <<~GQL
query {
  testField
}
GQL

result = RepoHeroSchema.execute(operation)
puts JSON.pretty_generate(result)
```

In this small GraphQL example, we start inside the `operation` string by defining the type of operation: a `query`.

Then inside that operation, we select fields. GraphQL operations are all about the *fields*. When we use a field in a GraphQL operation, we're telling the API that we want whatever data is defined for that field.

This script defines the GraphQL operation and will run it on the provided schema. We don't need the added complexity of what was in the controller — the variables, context or even a name for the operation — we just need the query.

Let's run this script now. We'll need it to load the `RepoHeroSchema`, and for that reason we'll run this script not using the `ruby` executable, but instead with `rails runner`:

```
rails runner graphql-test.rb
```

When we run the script, this is the output that we will see:

```
{
  "data": {
    "testField": "Hello World!"
  }
}
```

Our GraphQL API has returned us our first response. We requested a field called `testField`, and it returned us the data contained within that field, which is currently defined as the string `"Hello World!"`.

But how did our API know to return that value for that field? To get an answer to that, we'll need to dive into that `RepoHeroSchema` class and see how it fits together.

1.5. How the schema fits together

A GraphQL API is built around a schema, and a schema defines how the API behaves. Let's look at the schema that has been defined for our GraphQL API by that generator that we ran earlier.

Listing 4. app/graphql/repo_hero_schema.rb

```
class RepoHeroSchema < GraphQL::Schema
  mutation(Types::MutationType)
  query(Types::QueryType)
  ...
end
```

The main two types of operations, mutations and queries, are defined within their own type files. As a refresher: *mutations* are the operations that we use when we're creating, updating or deleting data. If we're reading data, we use *queries* instead.

We'll come back to mutations, but for now we will look at the `QueryType`.

Listing 5. app/graphql/query_type.rb

```
module Types
  class QueryType < Types::BaseObject
    ...

    # TODO: remove me
    field :test_field, String, null: false,
      description: "An example field added by the generator"
    def test_field
      "Hello World!"
    end
  end
end
```

The `QueryType` within our application defines a field and its response using the `field` method, and defining a method that matches the name of the field. But this is called `test_field`, and the field that we were requesting was called `testField`. So why the difference? Convention is why. GraphQL corresponds with JavaScript conventions, and JavaScript conventions are to favor `camelCase` names, over `snake_case` ones.

However, we're writing Ruby code in `QueryType`, and so the Ruby convention applies there and we use `test_field` instead. When in Ruby code, we'll use the Ruby conventions. The GraphQL gem will convert these to JavaScript conventions when appropriate.

After the name of the field in the `field` method are several other useful arguments.

The 2nd argument defines the type of the field's return value: a `String`.

The `null: false` declares that the return value for this field will never, ever be `null`, and so consumers of this API will not need to do any sort of null-checking on this field.

Finally, the `description` option contains documentation that would appear alongside this field in GraphiQL, or any other GraphQL documentation viewer.

If we were to change the return value of this method, we will see those changes reflected immediately. Just like we'd expect in any other piece of our Rails application code. Let's try this out now by changing the `test_field` method:

Listing 6. app/graphql/query_type.rb

```
def test_field
  "Hello GraphQL!"
end
```

Re-running our test script again:

```
rails runner graphql-test.rb
```

Will show that the output has changed:

```
{
  "data": {
    "testField": "Hello GraphQL!"
  }
}
```

We've now got a feel for how to use our GraphQL API, albeit through a script and not through the traditional request path. Now that we've seen how to use GraphQL on its own, let's integrate it with something we're familiar with from a Rails application: a model.

[1] RSpec has been chosen due to author preference. You could use Test::Unit for writing tests for GraphQL, if you wanted to.

[2] An explanation of why GraphQL uses POST: https://stackoverflow.com/questions/59162265/why-are-graphql-queries-post-requests-even-when-we-are-trying-to-fetch-data-and