GRAPHQL FOR RAILS DEVELOPERS

RYAN BIGG

GraphQL for Rails Developers

Build a GraphQL API for Your Rails App

Ryan Bigg

Table of Contents

Querying with GraphQL

We'll now add a model to track the repositories for Repo Hero. Repo Hero will be an application that people can use to review and rate their favorite repositories.

Since "repositories" is difficult for this author to type out so many times over and over, we're going to call the model "Repo" instead. Every repo will have a name and a URL. Let's run the generator now to create this model:

```
rails g model repo name:string url:string
```

This command will create the model, migration and the test files:

```
invoke  active_record
create    db/migrate/[timestamp]_create_repos.rb
create    app/models/repo.rb
invoke    rspec
create      spec/models/repo_spec.rb
```

Let's run the migration so that the `repos` table is created:

```
rails db:migrate
```

```
== [timestamp] CreateRepos: migrating
=====================================
-- create_table(:repos)
   -> 0.0008s
== [timestamp] CreateRepos: migrated (0.0008s)
============================
```

That's all the model setup that we need to handle right now. Next up, let's look at

how we can get GraphQL to read from this table using the model.

With our application setup to work with GraphQL, and a `Repo` model also in place, let's look at how we can use that model to fetch records from our database and display that information through GraphQL.

Listing repositories through GraphQL

To show a list of repositories from our `repos` table through the GraphQL API, we'll need to make some changes to our GraphQL code.

We'll approach solving this issue, and other issues throughout the book, by writing a test *first*, ensuring that it fails and then we'll write the code to make the test pass. This approach is chosen as it's going to be similar to the approach that would be expected of us if we were writing real-live production GraphQL code within a Rails application.

Tests for GraphQL code go into `spec/requests`, as we're going to be making requests straight to our application and then asserting on the response that we get back.

We'll write our first test into a new file at `spec/requests/graphql/queries/repos_spec.rb`:

Listing 1. spec/requests/graphql/queries/repos_spec.rb

```
require 'rails_helper'

RSpec.describe "Graphql, repos query" do
  let!(:repo) { Repo.create!(name: "Repo Hero", url:
"https://github.com/repohero/repohero") }
```

```
  it "retrieves a list of available repos" do
    query = <<~QUERY
    query {
      repos {
        name
        url
      }
    }
    QUERY

    post "/graphql", params: { query: query }
    expect(response.parsed_body["errors"]).to be_blank
    expect(response.parsed_body["data"]).to eq(
      "repos" => [
        {
          "name" => repo.name,
          "url" => repo.url,
        }
      ]
    )
  end
end
```

In this test we setup a repo that we should see being returned by our GraphQL API. Inside the test itself, we construct a query to retrieve a list of repos and their `name` and `url` fields. We then make a `POST` request to `/graphql` with our query, and expect there to be no errors, and to see the repo being returned here in the data.

When we run this test with `bundle exec rspec spec`, we'll see that it is currently failing:

```
Failure/Error: expect(response.parsed_body["errors"]).to be_blank
```

The output underneath that message is a little difficult to read. We could spelunk our way through it and see this message:

```
Field 'repos' doesn't exist on type 'Query'
```

Another way would be to take the `query` from our test, and to use it in our `graphql-test.rb` file:

```
query = <<~QUERY
query {
  repos {
    name
    url
  }
}
QUERY

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

We can then run this script with `rails runner graphql-test.rb`, and see exactly the same output our test is seeing:

```
{
  "errors": [
    {
      "message": "Field 'repos' doesn't exist on type 'Query'",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "query",
        "repos"
      ],
      "extensions": {
        "code": "undefinedField",
```

```
        "typeName": "Query",
        "fieldName": "repos"
      }
    }
  ]
}
```

We can get a similar kind of output in our test by defining a custom RSpec matcher. We'll do that now, and then we'll get right back to the GraphQL code, I promise.

This custom matcher's code goes into a new file at `spec/support/matchers/have_errors.rb`:

Listing 2. spec/support/matchers/have_errors.rb

```
RSpec::Matchers.define :have_errors do
  match do |response|
    response["errors"].present?
  end

  failure_message_when_negated do |response|
    "Expected there to be no errors, but there were:\n" +
      JSON.pretty_generate(response["errors"])
  end

  failure_message do |str|
    "Expected there to be errors, but there weren't"
  end
end
```

To ensure this file is loaded, we'll uncomment this line over in `spec/rails_helper.rb`:

```
Dir[Rails.root.join('spec', 'support', '**', '*.rb')].sort.each { |f|
require f }
```

To use the matcher, we can then change this line in our test:

Listing 3. spec/requests/graphql/queries/repos_spec.rb

```
expect(response.parsed_body["errors"]).to be_blank
```

To this:

```
expect(response.parsed_body).not_to have_errors
```

Doing this will mean that we'll now have a tidier output in our failing test output, and we won't have to chop-and-change between our tests and that `graphql-test.rb` file just to see a prettier version of our errors.

Let's run the test now with `bundle exec rspec` and see what happens. We will now see a much clearer failure message:

```
Failure/Error: expect(response.parsed_body).not_to have_errors

  Expected there to be no errors, but there were:
  [
    {
      "message": "Field 'repos' doesn't exist on type 'Query'",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "query",
        "repos"
      ],
      "extensions": {
        "code": "undefinedField",
        "typeName": "Query",
        "fieldName": "repos"
      }
    }

  ]
```

Much better! We can now see our error message much more clearly. Now how do we go about fixing this so that our test runs successfully? The hint is in the error message:

```
"Field 'repos' doesn't exist on type 'Query'"
```

This error message shows us that the query from the test is expecting there to be a field called `repos` on the type of `Query`. This `Query` type in GraphQL corresponds to the `app/graphql/query_type.rb` file that we saw earlier.

Listing 4. 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
```

Let's remove this `test_field` field from this class, but leave everything else. What we'll end up with is this:

Listing 5. app/graphql/query_type.rb

```
# frozen_string_literal: true

module Types
  class QueryType < Types::BaseObject
    field :node, Types::NodeType, null: true, description: "Fetches an
object given its ID." do
      argument :id, ID, required: true, description: "ID of the object."
    end

    def node(id:)
      context.schema.object_from_id(id, context)
    end

    field :nodes, [Types::NodeType, null: true], null: true, description:
"Fetches a list of objects given a list of IDs." do
      argument :ids, [ID], required: true, description: "IDs of the
objects."
    end
```

```
    def nodes(ids:)
      ids.map { |id| context.schema.object_from_id(id, context) }
    end
  end
end
```

Now we can add our `repos` field:

Listing 6. app/graphql/query_type.rb

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

    field :repos, [RepoType], null: false

    def repos
      Repo.all
    end
  end
end
```

With these changes, we're defining the `repos` field that our GraphQL query is expecting. The 2nd argument passed to `field` indicates to GraphQL that the type is going to be an array of `RepoType` objects. The `null: false` indicates that `repos` will never be null, at worst it'll instead be an empty array.

The `repos` method is the *resolver* for this field. When this `repos` field is accessed through a GraphQL operation, the GraphQL gem will use the `repos` method to *resolve* the data that will be displayed for that field. If we did not request the `repos` field, then the method would never be called.

This setup in `QueryType` is most of the way there. However, we've referenced a `RepoType` constant, but haven't defined it yet. This constant will define the fields that are accessible for Repo objects through the GraphQL API. Here's how we'll

define that type:

Listing 7. app/graphql/types/repo_type.rb

```
module Types
  class RepoType < Types::BaseObject
    field :name, String, null: false
    field :url, String, null: false
  end
end
```

In this type, we define two `String` fields, `name` and `url`, and both fields will never return `null` values. We do not need to define resolver methods for these fields, because the objects represented by this type already have `name` and `url` methods on them. The `graphql` gem will use those methods.

This will now be enough to get our test to pass. Let's run it and find out with `bundle exec rspec`:

```
1 example, 0 failures
```

Great! Let's recap what we've done.

To add a new field to our GraphQL API to read out a list of all of the repos, we used the `field` method within the `QueryType` class.

When we add a new field to the `QueryType`, we need to define how that field is resolved by the GraphQL API. To do that, we added a method matching the same name as the field to the `QueryType` class: `repos`. For this example, our `repos` method used `Repo.all`, which then returned an array of `Repo` model objects.

To work with these `Repo` objects in the GraphQL API, we needed to define another class called `RepoType`. This class defines how these `Repo` *model* objects are

represented in GraphQL. In this class we defined two fields: `name` and `url`. Importantly, we don't need to define resolver methods for these fields, as the `Repo` objects that `RepoType` represents already respond to the `name` and `url` methods, and so the `RepoType` class will use those methods from the `Repo` model class.

All of this, and we're now able to get a list of repos displaying in our API.

But what if we didn't want to display all of them at once? What if we wanted to display only a single repository's information? That's what we'll be looking at next.

Showing an individual repo

In a traditional Rails application in order to show information about a particular resource, you would define a route like:

```
GET /repos/:id
```

In GraphQL, we can build up a query and pass it a variable as well. The way we write it in GraphQL syntax is:

```
query ($id: ID!) {
  repo(id: $id) {
    name
    url
  }
}
```

The first line of our query now defines a *variable* (indicated by the dollar-sign) and its related type: `ID!`. This type takes in any record ID that we pass it, even if the ID was a string or a number. The exclamation-mark on the end of this type is GraphQL for "not null". Putting it all together: there's a variable called `$id` that accepts a value of type `ID`, and that value can be a string or a number, but never null.

On the 2nd line of the query, we retrieve the `repo` field and pass that `$id` variable as an *argument* to the field to indicate which repo we would like to fetch.

Now that we've looked at how to write that GraphQL query in theory, let's write a test and the associated code to make this a reality.

We'll create a new file at `spec/requests/graphql/queries/repo_spec.rb` and put this code into it:

Listing 8. spec/requests/graphql/queries/repo_spec.rb

```
require 'rails_helper'

RSpec.describe "Graphql, repo query" do
  let!(:repo) { Repo.create!(name: "Repo Hero", url:
"https://github.com/repohero/repohero") }

  it "retrieves a single repo" do
    query = <<~QUERY
    query ($id: ID!) {
      repo(id: $id) {
        name
        url
      }
    }
    QUERY

    post "/graphql", params: { query: query, variables: { id: repo.id } }
    expect(response.parsed_body).not_to have_errors
    expect(response.parsed_body["data"]).to eq(
      "repo" => {
        "name" => repo.name,
        "url" => repo.url,
      }
    )
  end
end
```

When we run this spec with `bundle exec rspec`, we'll be told that no such field exists:

```
Failure/Error: expect(response.parsed_body).not_to have_errors

  Expected there to be no errors, but there were:
  [
    {
      "message": "Field 'repo' doesn't exist on type 'Query'",
```

```
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "query",
        "repo"
      ],
      "extensions": {
        "code": "undefinedField",
        "typeName": "Query",
        "fieldName": "repo"
      }
    }
  ]
```

We know what to do here! We need to add a field to the `QueryType`. Let's jump over to `app/graphql/query_type.rb` and add that in.

When added the `repos` field earlier, we used the `field` method. We're going to use that same method again, but this time with a slight difference:

Listing 9. app/graphql/types/query_type.rb

```
field :repo, RepoType, null: false do
  argument :id, ID, required: true
end
```

We pass it a block this time! The block defines an *argument* for the field, specifying its type (`ID`), and that the argument is *required* for us to be able to resolve the field.

To define the resolver for this field, we'll add a new method called `repo` directly underneath this field definition:

```
def repo(id:)
  Repo.find(id)
end
```

This method takes in the `id` argument from the field, and uses it to resolve an object for GraphQL to use. That object is a single `Repo` instance. Our GraphQL API will use the `Repo` object here, and represent it through the `RepoType` class we defined earlier, because that is the type that we've defined on this new `repo` field.

When we run our spec again, we'll now see that it is passing:

```
2 examples, 0 failures
```

We now have a way of showing information about a single repository through our GraphQL API. To accomplish this, we allow for a `repo` field to be accessed through this query:

```
query ($id: ID!) {
  repo(id: $id) {
    name
    url
  }
}
```

This query is defined to take a variable called `$id`, which we then pass through as an argument to the `repo` field. Our API then takes this `id` argument and uses it in the resolver method in `QueryType` to find a `Repo` object, which is then represented in our API by the fields defined in the `RepoType` class.

We've hand-waved our way around how this `RepoType` class operates. This has been done so that we get familiar with how to define fields, and how to make those fields

take arguments. Now that we've got a grip on that foundational part of GraphQL, let's talk a little bit about how GraphQL knows how to use the `name` and `url` attributes from `Repo` objects. We'll walk through this process by adding another field.

Adding a custom field

We're now going to add a custom field to the `RepoType`. This field is going to be called `nameReversed`, and will return the `name` from a repo, but reversed. This means that "Repo Hero" will be returned as "oreH opeR" as the field's value.

To add a custom field to our `RepoType`, we need to use the `field` method again:

Listing 10. app/graphql/types/repo_type.rb

```
module Types
  class RepoType < Types::BaseObject
    field :name, String, null: false
    field :url, String, null: false
    field :name_reversed, String, null: false
  end
end
```

To access this field, we'll update our `repo_spec.rb` to fetch this field:

Listing 11. spec/requests/graphql/queries/repo_spec.rb

```
require 'rails_helper'

RSpec.describe "Graphql, repo query" do
  let!(:repo) { Repo.create!(name: "Repo Hero", url:
"https://github.com/repohero/repohero") }

  it "retrieves a single repo" do
    query = <<~QUERY
    query ($id: ID!) {
      repo(id: $id) {
        name
        nameReversed ①
        url
      }
    }
    QUERY

    post "/graphql", params: { query: query, variables: { id: repo.id } }
    expect(response.parsed_body).not_to have_errors
    expect(response.parsed_body["data"]).to eq(
      "repo" => {
        "name" => repo.name,
        "nameReversed" => repo.name.reverse, ②
        "url" => repo.url,
      }
    )
  end
end
```

① Adding the `nameReversed` field to the query

② Checking that the value comes back in the response

With these changes to both the `RepoType` and the test, let's see what happens when we run our test with `bundle exec rspec`:

```
RuntimeError:
```

```
  Failed to implement Repo.nameReversed, tried:

  - `Types::RepoType#name_reversed`, which did not exist
  - `Repo#name_reversed`, which did not exist
  - Looking up hash key `:name_reversed` or `"name_reversed"` on `#<Repo:<id>>`, but
it wasn't a Hash

  To implement this field, define one of the methods above (and check for typos), or
supply a `fallback_value`.
```

There's an error! This one even comes with a lot of explanation in the error message itself. It says that when GraphQL is attempting to resolve the `name_reversed` field, it does the following things in order:

1. Checks for a `name_reversed` instance method on `Types::RepoType`
2. Checks for a `name_reversed` instance method on the object being used for this field, a `Repo` instance
3. Checks to see if the `Repo` instance works as a `Hash`, attempting to access a `:name_reversed` or `"name_reversed"` key on that object.

Given that none of these things exist, we get this error.

We can solve this error by adding a method to `Types::RepoType` to handle this. The first instinct we might have here is to add this method:

Listing 12. app/graphql/types/repo_type.rb

```
def name_reversed
  name.reverse
end
```

As the `RepoType` already has a field called `name`, and so it seems logical that we should be able to access it by calling this method here too.

This will not work:

```
NameError:
      undefined local variable or method `name' for #<Types::RepoType:
```

The `RepoType` object, while it does have a *field* called `name`, it does not have a corresponding method called `name`.

In order to resolve values for fields, GraphQL uses another method called `object`. We can use it in our `name_reversed` method to get the behavior that we want:

Listing 13. app/graphql/types/repo_type.rb

```
def name_reversed
  object.name.reverse
end
```

The `object` method allows us to access the object that the `RepoType` is resolving. So when we use `RepoType` to resolve either the `repos` or `repo` field in our GraphQL API, the `object` that is resolved is an instance of the `Repo` model class.

By using the `object` method here to access the `Repo#name` method, we can then make our field do what our test is expecting it to do.

If we run our test at this point, we'll see that it is now passing:

```
2 examples, 0 failures
```

And there we have it! We've added a custom field to `RepoType` to return the reversed name. This part has demonstrated how we can use the `object` method to access the underlying object that is being used to resolve the fields of `RepoType`. This is most useful when we want to pre-format or modify values from the attributes on an

object, before they're presented through our GraphQL API.

One case where I use it like this is a fie|"ld called `formattedAmount` that takes a monetary amount and returns a formatted string. The `amount` field in the database there is a cents amount, for example `1234`, but I'd rather this be displayed nicely through the endpoint, so I have a `formattedAmount` field that then returns `"$12.34"` for that amount. That amount can then be displayed on the frontend without any additional formatting having to be applied.

Now that we've seen how to work with a single model within our application, let's look at how we can add a different model and query for that data through GraphQL.