Extracted from:

Craft GraphQL APIs in Elixir with Absinthe

Flexible, Robust Services for Queries, Mutations, and Subscriptions

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Craft GraphQL APIs in Elixir with Absinthe

Flexible, Robust Services for Queries, Mutations, and Subscriptions

Bruce Williams
Ben Wilson

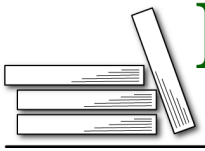
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Craft GraphQL APIs in Elixir with Absinthe

Flexible, Robust Services for Queries, Mutations, and Subscriptions

Bruce Williams
Ben Wilson

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Series Editor: Bruce A. Tate
Copy Editor: Nicole Abramowitz
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-255-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—April 2020

Making a Query

A GraphQL query is the way that API users can ask for specific pieces of information. We've defined the shape of our GraphQL MenuItem type, but to support users getting menu items, we need to provide two things:

- A way for users to request objects of the type
- A way for the system to retrieve (or resolve) the associated data

The key to the first objective is defining a special object type to serve as the entry point for queries on a GraphQL schema. We already defined it when we used the query macro earlier.

The query macro is just like object, but it handles some extra defaults for us that Absinthe expects. Since we've already defined a blank query object, let's take a look at what it looks like in IEx, too:

```
iex(1)> Absinthe.Schema.lookup_type(PlateSlateWeb.Schema, "RootQueryType")
```

The result looks something like this:

```
%Absinthe.Type.Object{
  identifier: :query,
  name: "RootQueryType",
  description: nil,
  fields: %{},
  interfaces: [],
  is_type_of: nil
}
```

As you can see, there's nothing special about the root query object type structurally. Absinthe will use it as the starting point of queries, determining what top-level fields are available.

Let's add the field we need, :menu_items, for our menu item listing query. We'll use the same field macro we used when we were building our :menu_item object:

```
query do

  field :menu_items, list_of(:menu_item)

end
```

list_of is a handy Absinthe macro that we can use to indicate that a field returns a list of a specific type. Technically, here it's shorthand for %Absinthe.Type.List{of_type: :menu_item}. That's a little long to type every time you need to return a list. We'll use menu_items, since it should return information about more than one menu item.

Supporting Language Conventions

GraphQL is often used by front-end languages like JavaScript that have slightly different conventions than Elixir. In Elixir, it's more conventional to use :menu_items, but in JavaScript, we'd expect menuItems (which is the GraphQL convention, as well).

Lucky for us, Absinthe handles translating between these two conventions automatically so that both the client and the server can work using the formats most familiar to them. The functionality is extensible, too; if you want to use a different naming convention in your GraphQL documents, you can.

Our :menu_items field doesn't actually build the list of menu items yet. To do that, we have to retrieve the data for the field. GraphQL refers to this as *resolution*, and it's done by defining a *resolver* for our field.

A field's resolver is the function that runs to retrieve the data needed for a particular field. Let's build our first one for the :menu_items field. Our menu item data is modeled using Ecto:[3]

02-chp.schema/2-object/lib/plate_slate/menu/item.ex

```
defmodule PlateSlate.Menu.Item do
  use Ecto.Schema
  import Ecto.Changeset
  alias PlateSlate.Menu.Item

  schema "items" do
    field :added_on, :date
    field :description, :string
    field :name, :string
    field :price, :decimal

    belongs_to :category, PlateSlate.Menu.Category

    many_to_many :tags, PlateSlate.Menu.ItemTag,
      join_through: "items_taggings"

    timestamps()
  end

  @doc false
  def changeset(%Item{} = item, attrs) do
    item
    |> cast(attrs, [:name, :description, :price, :added_on])
    |> validate_required([:name, :price])
    |> foreign_key_constraint(:category)
  end
end
```

3. https://hex.pm/packages/ecto

To retrieve all the menu items, do the following:

```
PlateSlate.Repo.all(PlateSlate.Menu.Item)
```

Since this is exactly what our :menu_items field needs to do, let's wire that in as the result of its resolver, using Elixir's alias to shorten the module names for readability:

02-chp.schema/2-object/lib/plate_slate_web/schema.ex

```
alias PlateSlate.{Menu, Repo}

query do

  field :menu_items, list_of(:menu_item) do
    resolve fn _, _, _ ->
      {:ok, Repo.all(Menu.Item)}
    end
  end

end
```

We've passed a function to the resolve macro to set the field's resolver. Because the field doesn't need any parameters, we can ignore the function arguments and just return an :ok tuple with the list of menu items. That lets Absinthe know that we were able to resolve the field successfully.

You don't need to define a resolver function for every field. For example, this query will attempt to resolve a menu item's :name field:

```
{
  menuItems {
    name
  }
}
```

If a resolver is not defined for a field, Absinthe will attempt to use the equivalent of Map.get/2 to retrieve a value from the parent value in scope, using the identifier for the field. You'll learn more about how that works in Setting Defaults, on page ?.

Resolution starts at the root of a document and works its way deeper, with each field resolver's return value acting as the parent value for its child fields. Because the resolver for menuItems (that is, the resolver we defined in our schema for the :menu_items field) returns a list of menu item values—and resolution is done for each item in a list—the parent value for the name field is a menu item value. Our query, in fact, boils down to something very close to this:

```
for menu_item <- PlateSlate.Repo.all(PlateSlate.Menu.Item) do
  Map.get(menu_item, :name)
```

```
end
```

Of course, our GraphQL request gets this information bundled up, nicely labeled in a JSON response from Absinthe.

Let's take a break from editing the schema to play with GraphiQL, a handy user interface we can use to query our fledgling GraphQL API.

Running Our Query with GraphiQL

GraphiQL is "an in-browser IDE for exploring GraphQL," and to make things easy for the user, Absinthe integrates with three versions of GraphiQL: the official interface,[4] an advanced version,[5] and GraphQL Playground.[6] All three are built in to the absinthe_plug[7] package and ready to go with just a little configuration.

The absinthe_plug dependency is already in our mix.exs file from the initial setup, but we need to now configure the Phoenix router to use it. Replace the existing "/" scope with the following block:

02-chp.schema/2-object/lib/plate_slate_web/router.ex

```
scope "/" do
  pipe_through :api

  forward "/api", Absinthe.Plug,
    schema: PlateSlateWeb.Schema
```

4. https://github.com/graphql/graphiql
5. https://github.com/OlegIlyenko/graphiql-workspace
6. https://github.com/graphcool/graphql-playground
7. https://github.com/absinthe-graphql/absinthe_plug

```
  forward "/graphiql", Absinthe.Plug.GraphiQL,
    schema: PlateSlateWeb.Schema,
    interface: :simple
end
```

Really, we're setting up two routes: "/api" with the regular Absinthe.Plug, and "/graphiql" with the GraphiQL plug. The former is what API clients would use and what we'll use in our tests, and then the latter provides the "in-browser" IDE we'll use now. Specifically, we're going to use the simplified, official GraphiQL interface, set with the interface: :simple option.

Let's start our application by running the following:

```
$ mix phx.server
```

Since the server will start on port 4000, visit http://localhost:4000/graphiql (adding the path where you have mounted GraphiQL) and see the GraphiQL user interface.

There's a lot to see here, but let's give our query a shot before we dig into it much further. Start by typing your query into the text area to the top left.

Did you notice that while you were typing, GraphiQL helpfully suggested some autocompletions? That's because when you loaded the page, it automatically sent an introspection query to your GraphQL API, retrieving the metadata it needs about PlateSlateWeb.Schema to support autocompletion and display documentation.

When you press the play button above the query, you can see the JSON result in the right-hand text area as shown in the top figure on page 10.

Success! Now, let's try this one, adding the :id field:

```
{
  menuItems {
    id
    name
  }
}
```

Here's the result:

It's handy being able to specify additional fields in our query without having to modify the schema any further! We already defined the :id field on our :menu_item type, so it works out of the box. We just weren't asking for it before.

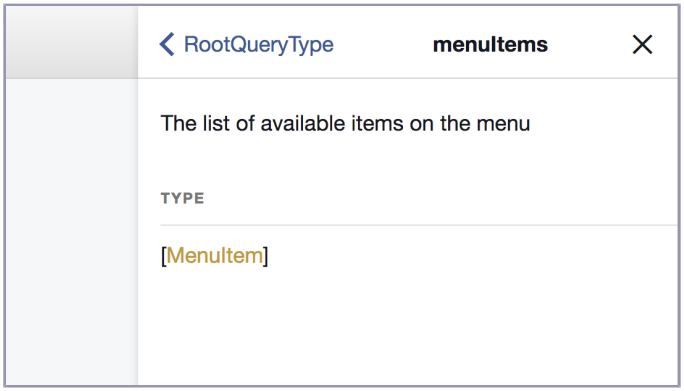
What else can we query? Let's look at the API documentation that GraphiQL has collected for us. To the right of the GraphiQL interface, there's a "Docs" link that, when clicked, will open up a new sidebar full of API documentation:

If you click on RootQueryType, you can see the menuItems field with its type, [MenuItem], displayed, but it's missing a more detailed description. You can add one by editing your schema.

Let's do that now. Back in web/schema.ex, you can add a :description value as part of the third argument to the field macro:

```
field :menu_items, list_of(:menu_item),
      description: "The list of available items on the menu" do
  «Menu item field definition»
end
```

If you look back at GraphiQL (refresh the page), your description will now be displayed.



There's another technique you can use to add descriptions, using a module attribute, @desc, just as you would with Elixir's @doc:

```
@desc "The list of available items on the menu"
field :menu_items, list_of(:menu_item) do
  «Menu item field definition»
end
```

Because the latter approach supports multiline documentation more cleanly and sets itself off from the working details of our field definitions, it's the approach we'll use in our application.

Testing Our Query

GraphiQL is a great tool to explore our API and when we'd like to manually run a query, but it's not a replacement for a test suite. We'll use *ExUnit* to add tests for our Absinthe schema to make sure our queries work now and later on to prevent regressions. Our future selves will appreciate the forethought.

ExUnit is bundled with Elixir, so no dependencies are required. Since our PlateSlate application is using Phoenix, ExUnit has already been set up with a preconfigured test harness that we can use.

Because we know our users are going to use the API by hitting /api, we can treat our API just as we would a Phoenix controller, using the PlateSlate.ConnCase helper module that Phoenix generously generated for us:

02-chp.schema/2-object/test/plate_slate_web/schema/query/menu_items_test.exs

```
defmodule PlateSlateWeb.Schema.Query.MenuItemsTest do
  use PlateSlateWeb.ConnCase, async: true

  setup do
    PlateSlate.Seeds.run()
  end

  @query """
  {
    menuItems {
      name
    }
  }
  """
  test "menuItems field returns menu items" do
    conn = build_conn()
    conn = get conn, "/api", query: @query
    assert json_response(conn, 200) == %{
      "data" => %{
        "menuItems" => [
          %{"name" => "Reuben"},
          %{"name" => "Croque Monsieur"},
          %{"name" => "Muffuletta"},
          # «Rest of items»
        ]
      }
    }
  end

end
```

The `setup` block loads our seed data as a convenience. The test itself starts by building a connection. Then it passes the `@query` module attribute we defined previously (making use of Elixir's handy multiline `"""` string literal) as the `:query` option, which is what `Absinthe.Plug` expects. The response is then checked to make sure that it has an HTTP `200` status code and includes the JSON data that we expect to see.

Running the test gives us exactly what we were hoping for:

```
$ mix test test/plate_slate_web/schema/query/menu_items_test.exs
.

Finished in 0.2 seconds
1 test, 0 failures
```

After a little compilation, a passing test!

We'll continue to build tests out this way as our API grows. These tests exercise a lot of our system, from HTTP requests through JSON serialization, helping

to reduce our stress by keeping us confident that changes elsewhere in the application aren't affecting our GraphQL users.

Moving On

In this chapter, we learned how to build the foundation of a GraphQL schema in an Elixir application, defining an object type that we exposed via a query field, and we tested our fledgling API using a popular tool, GraphiQL.

Here are a couple challenges for you before we move on:

1. We've defined `:id` and `:name` fields for our `MenuItem` object type. The backing Ecto schema, `PlateSlate.Menu.Item`, has a number of other fields we could also expose in our GraphQL schema. Define another one using one of the built-in scalar types we mentioned earlier.
2. Add descriptions for the fields inside the `MenuItem` object type, using the `@desc` form. Don't stop there: you can use it to add a description for the object type itself, too. Verify that GraphiQL is displaying the descriptions.

Once you're done, we're going to look at supporting user input in the next chapter, which will open up a whole range of new and interesting API possibilities.