PHOENIX 1.3 AND GRAPHQL WITH ABSINTHE

APR 18, 2017

Phoenix and **GraphQL** via **Absinthe** is an extremely liberating experience when developing APIs. **Phoenix 1.3** has introduced a new (optional) file structure and new concepts that aren't present in Phoenix <= 1.2. Absinthe works great with these changes, but it currently takes some guesswork to do that. I will go through the process to do that in this article.

Problems with Phoenix 1.3 and Absinthe

Phoenix 1.3 Changes

Phoenix 1.3 has added some new concepts and removed some concepts. If you already are aware of these changes, feel free to skip to the next section. The overall changes have been overviewed by Chris McCord over the span of a few conference talks that you can see below:

Lonestar ElixirConf 2017- KEYNOTE: Phoenix 1.3 by Chris McCord

Lonestar ElxirConf 2017 Keynote

ElixirConf 2016 Keynote (Bad audio—sorry!)

Absinthe

Absinthe is an Elixir GraphQL implementation specifically-made for Phoenix. Unfortunately, Phoenix 1.3 (as of this writing) is only in the Release Candidate stage, and the documentation/tutorials you'll find for Absinthe will mostly only cover Phoenix 1.2 and don't cover the 1.3 changes. For the record: Absinthe works *perfectly* fine with Phoenix 1.3, you just have to do some digging around and guessing to fit it in with 1.3 contexts and directory structure. Most tutorials still reference models and such that no longer exist in new 1.3 projects, and that's what we're going to cover today.

Our Project

The project we'll be making is solely a GraphQL API that fetches blog posts and user accounts. Users own blog posts and blog posts are owned by users. It is a *very* simple API since it is only meant to cover the basics to get you started using Absinthe with Phoenix 1.3 projects. If you wish to go deeper, nearly all Absinthe tutorials you find can carry over using the information laid out from here.

This project will not go in-depth on GraphQL, Phoenix or Absinthe. I'm going to assume you know how to set up Absinthe in a Phoenix 1.2 application. It will show you how to make Absinthe work with 1.3 and how it differs with 1.2. There are plenty of smarter people than I who can tell you all about these projects.

Pre-Requisites

- Phoenix 1.3 installed. Instructions for that can be found here: https://gist.github.com/chrismccord/71ab10d433c98b714b75c886eff17357
- PostgreSQL installed. Instructions here: https://wiki.postgresql.org/wiki/Detailed*installation*guides

Generating a New Phoenix 1.3 Project

Phoenix 1.3 comes with a brand new suite of `mix` tasks. All of the `mix phoenix` tasks are *deprecated* and most likely will be removed in Phoenix 2.0 (that's pure speculation, though). These are scoped to `mix phx`. To create a new Phoenix 1.3 project, just run `mix phx.new [APP_NAME]` like you used to run `mix phoenix.new [APP_NAME]`. For the purposes of this project, I'm running it with the `--no-html` and `--no-brunch` flags. Here's what I'm running:

```
mix phx.new blog_app --no-html --no-brunch
```

So follow the Phoenix instructions and grab your deps, compile, set up ecto, etc. If we look at `blog_app/lib` after all that is done, it should look something like this:

```
lib
└── blog_app
    ├── application.ex
    ├── repo.ex
    └── web
        ├── channels
        |   └── user_socket.ex
        ├── controllers
        ├── endpoint.ex
        ├── gettext.ex
        ├── router.ex
        ├── views
        |   ├── error_helpers.ex
        |   └── error_view.ex
        └── web.ex
```

You'll notice the `web` folder is now inside `lib` instead of being a sibling of it!

Creating `users` and `posts`

So we're going to have two different contexts: An `Accounts` context and a `Blog` context. The Accounts context will take care of our `users`, while the Blog context will cover our `posts`.

Let's generate these with the new `mix phx.gen` generators!

For our User, we'll have a `name` that is of type `string`, as well as an `email` with the same type.

```
mix phx.gen.json Accounts User users name:string email:string
```

For our Post, we'll have a `title` of type string, a `body` of type `text`, as well as a reference to the `accounts_users` table that was created in the previous step.

```
mix phx.gen.json Blog Post posts title:string body:text accounts_users_id:references:accounts_use
```

Make sure you follow the steps about adding resources/etc that Phoenix generates. They should read somewhat like this:

```
Add the resource to your api scope in lib/blog_app/web/router.ex:

    resources "/users", UserController, except: [:new, :edit]

Remember to update your repository by running migrations:

    $ mix ecto.migrate


Add the resource to your api scope in lib/blog_app/web/router.ex:

    resources "/posts", PostController, except: [:new, :edit]

Remember to update your repository by running migrations:

    $ mix ecto.migrate
```

Our directory structure should now look something like this:

```
lib
└── blog_app
    ├── application.ex
    ├── repo.ex
    ├──accounts
    |   ├── accounts.ex # Context
    |   └── user.ex     # Schema
    ├── blog
    |   ├── blog.ex     # Context
    |   └── post.ex     # Schema
    └── web
        ├──channels
        |   └── user_socket.ex
        ├──controllers
        |   ├── fallback_controller.ex
        |   ├── post_controller.ex
        |   └── user_controller.ex
        ├── endpoint.ex
        ├── gettext.ex
        ├── router.ex
        ├──views
        |     views
        |    ├── changeset_view.ex
        |    ├── error_helpers.ex
        |    ├── error_view.ex
        |    ├── post_view.ex
        |    └── user_view.ex
        └───web.ex
```

Modifications

Just like in 1.2, you have to modifiy your schemas in 1.3.

Here's the diff of my `user.ex` in `accounts`:

```
  schema "accounts_users" do
    field :email, :string
    field :name, :string
+ has_many :blog_posts, BlogApp.Blog.Post, foreign_key: :accounts_users_id

    timestamps()
  end
```

And with `post.ex` in `blog`:

```
  schema "blog_posts" do
    field :body, :string
    field :title, :string
+ belongs_to :accounts_users, BlogApp.Accounts.User, foreign_key: :accounts_users_id

    timestamps()
  end
```

Adding Absinthe

Now to add Absinthe to our project. First, let's add some deps to our `mix.exs` file.

```
  defp deps do
-     [{:phoenix, "~> 1.3.0-rc"},
-      {:phoenix_pubsub, "~> 1.0"},
-      {:phoenix_ecto, "~> 3.2"},
-      {:postgrex, ">= 0.0.0"},
-      {:gettext, "~> 0.11"},
-      {:cowboy, "~> 1.0"}]
+     [
+       {:phoenix, "~> 1.3.0-rc"},
+       {:phoenix_pubsub, "~> 1.0"},
+       {:phoenix_ecto, "~> 3.2"},
+       {:postgrex, ">= 0.0.0"},
+       {:gettext, "~> 0.11"},
+       {:cowboy, "~> 1.0"},
+       {:absinthe, "~> 1.3.0-rc.0"},
+       {:absinthe_plug, "~> 1.3.0-rc.0"},
+       {:absinthe_ecto, git: "https://github.com/absinthe-graphql/absinthe_ecto.git"},
+       {:faker, "~> 0.7"},
+     ]
  end
```

For the most part, all of that is self-explanatory. You *should* be able to use version `1.2` of absinthe and absinthe_plug, but I'm sticking with the new just because. I added `faker` in there for seeding the db with some data. More on that later.

Adding GraphQL Endpoints

I added my GraphQL endpoints like this in `lib/[APP]/web/router.ex` just like a Phoenix 1.2 application.

```
    ...
      resources "/posts", PostController, except: [:new, :edit]
    end
  +
  +  forward "/graph", Absinthe.Plug,
  +    schema: BlogApp.Schema
  +
  +  forward "/graphiql", Absinthe.Plug.GraphiQL,
  +    schema: BlogApp.Schema
  end
```

Defining Schema Types

Your types should go in the `schema` folder in `web`. just like it did before in 1.2. Remember that the `web` folder is now in `lib`. This isn't required, obviously, because it's Elixir and the module-resolution system is robust, but this is to follow convention for Absinthe applications.

Mine will look like this:

```
  defmodule BlogApp.Schema.Types do
    use Absinthe.Schema.Notation
    use Absinthe.Ecto, repo: BlogApp.Repo

    object :accounts_user do
      field :id, :id
      field :name, :string
      field :email, :string
      #
      # Take note on the names here:
      # list_of(:blog_post) -> :blog_post maps to the Absinthe object down below
      # while assoc(:blog_posts) maps to the table!
      # The table names are named that because of Phoenix contexts that we made earlier!
      #
      field :posts, list_of(:blog_post), resolve: assoc(:blog_posts)
    end

    object :blog_post do
      field :id, :id
      field :title, :string
      field :body, :string
      #
      # You can see the same pattern here:
      # field :user, :accounts_user -> :accounts_user = the object above
      # assoc(:accounts_users) -> :accounts_users = the accounts_users table
      #
      field :user, :accounts_user, resolve: assoc(:accounts_users)
    end
  end
```

Defining Schemas

Again, as convention goes, the schema should go in `lib/[APP]/web/schema.ex`. Here's mine:

```
  defmodule BlogApp.Schema do
    use Absinthe.Schema
    import_types BlogApp.Schema.Types

    query do
      field :blog_posts, list_of(:blog_post) do
        resolve &BlogApp.Blog.PostResolver.all/2
      end
      field :accounts_users, list_of(:accounts_user) do
        resolve &BlogApp.Accounts.UserResolver.all/2
      end

    end
  end
```

Creating Resolvers

Resolvers are a little different than in 1.2, only because of the new folder structure in 1.3. I think it's best to keep the resolvers as siblings to your schemas in `lib/[APP]/[CONTEXT]/`. For example, my `PostResolver` will go in `lib/blog_app/blog/post_resolver.ex`, a sibling to `post.ex`.

Here's my resolver:

```
  defmodule BlogApp.Blog.PostResolver do
    alias BlogApp.{Blog.Post, Repo}

    def all(_args, _info) do
      {:ok, Repo.all(Post)}
    end
  end
```

Nothing special, looks just like a regular ol' resolver in 1.2.

My `UserResolver` is the same: a sibling to `user.ex` in my `accounts` context. It lives at `lib/blog_app/accounts/user_resolver.ex` for me.

Here's that resolver:

```
  defmodule BlogApp.Accounts.UserResolver do
    alias BlogApp.{Accounts.User, Repo}

    def all(_args, _info) do
      {:ok, Repo.all(User)}
    end
  end
```

Seeding Data

I seeded my db with this seed file. It generates 10 users, then generates 40 posts and assigns them to a random user:

```
  alias BlogApp.Repo
  alias BlogApp.Accounts.User
  alias BlogApp.Blog.Post

  # Create 10 seed users

  for _ <- 1..10 do
    Repo.insert!(%User{
      name: Faker.Name.name,
      email: Faker.Internet.safe_email
    })
  end

  # Create 40 seed posts

  for _ <- 1..40 do
    Repo.insert!(%Post{
      title: Faker.Lorem.sentence,
      body: Faker.Lorem.sentences(%Range{first: 1, last: 3}) |> Enum.join("\n\n"),
      accounts_users_id: Enum.random(1..10) # Pick random user for post to belong to
    })
end
```

Wrap-Up

If everything went well we should be able to run `mix phx.server`, hit `localhost:4000/graphiql` and start running some sweet sweet GraphQL queries!

Remember that Phoenix 1.3 namespaces a lot of things with the new "context" concept. Tables, modules, etc all take in after it and we have to keep that in mind when reading educational material referencing 1.2. Phoenix 1.3 also removes all mentions of the word "model" so we have to map that to 1.3 schemas.

If you'd like to take at the complete repo, take a look-see here: **https://github.com/sean-clayton/blog_app**