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

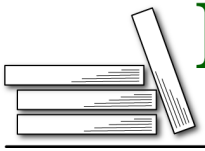
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

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

Web APIs need to define and validate input from users, whether it's used to query information or to modify it. In most web frameworks, this input definition is ad hoc and often mixed in with the business logic of the application. GraphQL takes a more declarative approach, however, by defining input as part of your API schema and supporting type validations as a core feature.

In this chapter, you'll see that by articulating the rules about our data in the schema, we can have the Absinthe package enforce them for us, allowing our Elixir application code to focus on more core application concerns. This will make our code more readable and easier to maintain.

We'll dig into the nuts and bolts of user input in GraphQL, covering the different ways users can provide it and the constraints we can set. You'll learn about new input types and how to apply them to make your GraphQL schemas more descriptive, accurate representations of your API.

Let's start by looking at GraphQL's most fundamental user input concept, the field argument.

Defining Field Arguments

GraphQL documents are made up of fields. The user lists the fields they would like, and the schema uses its definition of those fields to resolve the pieces of data that match. The system would be pretty inflexible if it did not also allow users to provide additional parameters that would clarify exactly what information each field needed to find. A user requesting information about `menuItems`, for instance, may want to see certain menu items or a certain number of them.

It's for this reason that GraphQL has the concept of field *arguments*: a way for users to provide input to fields that can be used to parameterize their queries. Let's take a look at our example application and see how we can extend our Absinthe schema by defining the arguments that our API will accept for a field, and then see how we can use those arguments to tailor the result for users.

We've already built a field in our API that we could make more flexible by accepting user input: the list of menu items. Our schema's `menuItems` field, if you remember, looks something like this:

03-chp.userinput/1-start/lib/plate_slate_web/schema.ex

```
Line 1  alias PlateSlate.{Menu, Repo}
-
-  query do
-
5    field :menu_items, list_of(:menu_item) do
```

```
    resolve fn _, _, _ ->
      {:ok, Repo.all(Menu.Item)}
    end
  end

end
```

On line 7, the field's resolver just returns all the menu items, without any support for filtering, ordering, or other modifications to the scope or layout of the result. The field isn't declaring any arguments, so the resolver doesn't receive anything with which we could modify the list of menu items retrieved.

Let's add an argument to our schema to support filtering menu items by name. We'll call it matching, then configure our field resolver to use it when provided:

03-chp.userinput/2-matchinginline/lib/plate_slate_web/schema.ex

```
Line 1 alias PlateSlate.{Menu, Repo}
import Ecto.Query

query do

  field :menu_items, list_of(:menu_item) do
    arg :matching, :string
    resolve fn
      _, %{matching: name}, _ when is_binary(name) ->
        query = from t in Menu.Item, where: ilike(t.name, ^"%#{name}%")
        {:ok, Repo.all(query)}
      _, _, _ ->
        {:ok, Repo.all(Menu.Item)}
    end
  end

end
```

On line 7, we defined matching as a :string type. If you remember from the previous chapter, :string is a built-in type. We can use it as an input type, too.

We're not making the matching argument mandatory here, so we need to support resolving our menuItems field in the event it's provided, and in the event it isn't. You can see Elixir's pattern matching capability used in the two separate function heads of our resolver to handle those two cases.

The second function head, on line 12, serves as the fall-through match and is identical to our original resolver.

It's the first function head, on line 9, that adds our new behavior. On line 10, we make use of the matched argument as name (in the from macro that Ecto.Query[1] provides) to build our Ecto query. We pulled the Ecto.Query macros in on line 2. By declaring our inputs up front, Absinthe has a bounded set of inputs to work with and can thus give us an atom-keyed map to work with as arguments, unlike Phoenix controller action params.

Resolvers and Field Arguments

Absinthe only passes arguments to resolvers if they have been provided by the user. Making a map key match of the arguments resolver function parameter is a handy way to check for a field argument that's been specified in the request.

Writing complicated resolvers as anonymous functions can have a negative side effect on a schema's readability, so to keep the declarative look and feel of the schema alive and well, let's do a little refactoring and extract the resolver into a new module. Because filtering menu items is an important feature of our application—and could be used generally, not just from the GraphQL API—we'll also pull the core filtering logic into the PlateSlate.Menu module, which is where our business logic relating to the menu belongs.

Here's our new resolver module:

03-chp.userinput/3-matching/lib/plate_slate_web/resolvers/menu.ex

```
defmodule PlateSlateWeb.Resolvers.Menu do
  alias PlateSlate.Menu

  def menu_items(_, args, _) do
    {:ok, Menu.list_items(args)}
  end
end
```

You can see that the resolver is calling PlateSlate.Menu.list_items/1, passing the arguments. The logic inside PlateSlate.Menu looks like this:

03-chp.userinput/3-matching/lib/plate_slate/menu/menu.ex

```
def list_items(%{matching: name}) when is_binary(name) do
  Item
  |> where([m], ilike(m.name, ^"%#{name}%"))
  |> Repo.all
end
def list_items(_) do
  Repo.all(Item)
end
```

1. https://hexdocs.pm/ecto/Ecto.Query.API.html

This code should look pretty familiar; it's been extracted out of our anonymous resolver function and restructured into a named function. Doing this makes both the resolver and the overall schema more readable.

The Point of "Pointless" Modules

While it might seem like adding resolver modules just to have them call functions from other modules is superfluous, it's important to set up a solid separation of concerns early on in our project.

In general, a resolver's job is to mediate between the input that a user sends to our GraphQL API and the business logic that needs to be called to service their request. As your schema gets more complex, you'll be glad you made space in the overall architecture of your application to keep your resolver and domain business logic separate.

We'll cover structural decisions like these in more detail in Chapter 4, Adding Flexibility, on page ?.

Now let's wire our resolver back into our :menu_items field in the schema:

03-chp.userinput/3-matching/lib/plate_slate_web/schema.ex

```
alias PlateSlateWeb.Resolvers

query do

  field :menu_items, list_of(:menu_item) do
    arg :matching, :string
    resolve &Resolvers.Menu.menu_items/3
  end

end
```

Using Elixir's & function capture special form[2] here lets us tie in the function from our new module as the resolver for the field and keeps the schema declaration tight and focused.

Let's explore using this new field argument that we've defined with some GraphQL queries that cover a range of scenarios.

Providing Field Argument Values

There are two ways that a GraphQL user can provide argument values for an argument: as document literals, and as variables.

2. https://hexdocs.pm/elixir/Kernel.SpecialForms.html#&/1

Using Literals

Using document literals, values are embedded directly inside the GraphQL document. It's a straightforward approach that works well for static documents. Here's a query that uses a document literal for the matching argument that we've added to retrieve menu items whose names match "reu":

```
{
  menuItems(matching: "reu") {
    name
  }
}
```

Argument values are given after the argument name and a colon (:), and the literal for a :string argument is enclosed in double quotes ("). Let's use this query in a new test, just as we did in Testing Our Query, on page ?:

03-chp.userinput/3-matching/test/plate_slate_web/schema/query/menu_items_test.exs

```
@query """
{
  menuItems(matching: "reu") {
    name
  }
}
"""
test "menuItems field returns menu items filtered by name" do
  response = get(build_conn(), "/api", query: @query)
  assert json_response(response, 200) == %{
    "data" => %{
      "menuItems" => [
        %{"name" => "Reuben"},
      ]
    }
  }
end
```

Running the test, we can verify that our literal argument value is being passed through, and the query successfully filtering the menu items returned:

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

Finished in 0.4 seconds
2 tests, 0 failures
```

It works! Now let's see what happens when a user provides a bad value:

03-chp.userinput/3-matching/test/plate_slate_web/schema/query/menu_items_test.exs

```
@query """
{
  menuItems(matching: 123) {
```

```
      name
    }
  }
  """
  test "menuItems field returns errors when using a bad value" do
    response = get(build_conn(), "/api", query: @query)
    assert %{"errors" => [
      %{"message" => message}
    ]} = json_response(response, 400)
    assert message == "Argument \"matching\" has invalid value 123."
  end
```

The first thing to notice here is that we're getting an HTTP 400 response code from Absinthe. This indicates that one or more errors occurred that prevented query execution. Helpfully, the error given in the response tells the user of the API what they're doing wrong.

This is great! Our API can respond appropriately to user-provided values, without any intervention by any custom type-checking code. By consulting our schema, Absinthe handles it for us.

Let's run it to make sure the error is returned:

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

Finished in 0.4 seconds
3 tests, 0 failures
```

Now that we have both valid and invalid tests working correctly, let's talk about how users might use this query in the real world. In the examples we're using for our tests, we're using literal argument values directly in the GraphQL document. This isn't very reusable.

Imagine a user interface that took a search term from end users and then called out to our API. If the front-end application only used document literals, it would need to interpolate the search terms directly into the GraphQL document. For each user request, a completely new document would have to be generated, likely using string interpolation. To do this while ensuring that the GraphQL document wouldn't be malformed, it would need to sanitize the input—making sure, for instance, that no double quotes were provided that would prematurely end the string value and cause a parse error from the GraphQL server.

This is a great use case for GraphQL *variables*—a way to insert dynamic argument values provided alongside (rather than inside) the static GraphQL document.

Using Variables

GraphQL variables act as typed placeholders for values that will be sent along with the request, a concept that may be familiar to you if you've used parameterized SQL queries for insertion and sanitization of values. GraphQL variables are declared with their types—before they're used—alongside the *operation* type. We haven't had to think about operation types before, so let's talk a little bit about what operations are and how they fit inside the GraphQL document.

Understanding Operations

A GraphQL document consists of one or more operations, which model something that we want the GraphQL server to do. Up to this point, we've been asking the server to provide information—an operation that GraphQL calls a `query`. GraphQL has other operation types too, notably `mutation` for persisting a change to data, and `subscription` to request a live feed of data. We'll get into those later.

We've been using a simplified way of typing up a *query operation*, which just uses an outer set of curly braces ({}) to demarcate where it starts and ends:

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

GraphQL assumes that if you're providing a single operation like this, its operation type is `query`. The previous example is equivalent to this, where we explicitly mark the operation as a query:

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

In simple cases, we omit the operation type, but when we're using variables, we need to use the more formal, verbose syntax and fully declare the operation. This gives us a place to list and describe the variables that we'll be using in the operation. Let's declare a variable for use in our menu item search query.

Naming Operations

You can also provide a name for operations, which can be useful for identifying them in server logs. The name is provided after the operation type—for instance, `query MenuItemList { ... }`. You'll see named operations later in the book.

Declaring Variables

Here's our menu items query operation with a definition for a variable we'll be using, `$term`, and then its use for the `matching` argument:

```
query ($term: String) {
  menuItems(matching: $term) {
    name
  }
}
```

Variable declarations are provided directly before the curly braces that start the body of an operation, and are placed inside a set of parentheses. Variable names start with a dollar sign (`$`), and their GraphQL types follow after a colon (`:`) and a space character. If we were declaring multiple variables, we'd list them separated by commas.

The variable's GraphQL type isn't the `snake_cased` form as declared in our schema. As you discovered in the previous chapter, Absinthe uses `snake_cased` atom identifiers for GraphQL types (like `:string`) so that our Elixir code feels idiomatic. In GraphQL documents, however, we need to use the canonical GraphQL type names (like `String`), which are `CamelCased`. (If you're ever unsure of the canonical name for a built-in GraphQL type and how they map to Absinthe types, take a peek at Appendix 1, GraphQL Types, on page ?, where we've laid them all out for you.)

We used the `String` type for our `$term` variable, since that's exactly the type of argument value that we defined for the `matching` argument in our schema.

Variable Types Aren't Extraneous

While it might seem like having to declare an argument type and a variable type (that will be used for that argument) is overkill, it allows the GraphQL server to give clearer error messages about the expected vs. provided variable value and lets the GraphQL document writer make values mandatory to support client-side validation.

Of course, if you've taken the time and effort to declare a variable and sprinkle its values throughout a document, you probably want to know how to provide values for it.