Server Side GraphQL

Using Elixir, Phoenix and Absinthe

Robert Boone

ChaiOne

June 27, 2018

Table of Contents

Elixir

Elixir is a functional, concurrent, general-purpose programming language that runs on the Erlang virtual machine (BEAM). Elixir builds on top of Erlang and shares the same abstractions for building distributed, fault-tolerant applications. Elixir also provides a productive tooling and an extensible design. The latter is supported by compile-time metaprogramming with macros and polymorphism via protocols.

Syntax

- Pipe Operator
- With Macro
- Modules and Functions
- Pattern Matching
- Enum Module
- For Comprehension
- Use

Pipe Operator

- One aspect of functional programming is composing functions.

Pipe Operator

- One aspect of functional programming is composing functions.
- You compose functions by passing the output of one function as the input of the next.

Pipe Operator

- One aspect of functional programming is composing functions.
- You compose functions by passing the output of one function as the input of the next.
- By doing this you can make pipe lines of functions that transforms the data that is passed to it.

Pipe Operator

Without the Pipe Operator function composition can look like this:

```
four(three(two(one(x), global: false)))

x1 = one(x)
x2 = two(x1, global: false)
x3 = three(x2)
x4 = four(x3)
```

Pipe Operator

```
String.upcase("hello world")
"HELLO WORLD"

"hello world" |> String.upcase()
"HELLO WORLD"
```

Pipe Operator

```
x
|> one()
|> two(global: false)
|> three()
|> four()
```

Modules and Functions

```elixir
defmodule Core do
  def add(a,b) do
    a + b
  end
end

Core.add(1,1) == 2

add = fn(a,b) -> a + b end
add.(1,1) == 2
```

Listing 6: Module

Pattern Matching

```elixir
defmodule Core do
  def email(%User{name: name, email: email} = user) do
    IO.puts("#{name}'s email is #{email}.")
    user
  end

  def email(%Admin{} = admin) do
    IO.puts("admin@example.com")
    admin
  end
end

email =
  fn
    (%User{email: email} = user) -> "#{user.name}'s email is #{email}"
    (%Admin{})                    -> "admin@example.com"
  end

email.(user)
```

Listing 7: Pattern Matching

Enum Module

```
Enum.__info__(:functions) |> length
96
```

Common Functions

- each
- map
- filter
- reduce

With Macro

Another way to tackle function composition:

```
def login(_obj, args, _ctx) do
  user = Repo.get_by(User, email: args.email)

  with {:ok, user} <- Comeonin.Argon2.check_pass(user, args.password),
       {:ok, token, _claims} <- Fari.Guardian.encode_and_sign(user) do
    {:ok, %{token: token}}
  else
    {:error, "invalid password"} ->
      {:error, "Bad Username or password"}

    {:error, message} ->
      {:error, message}
  end
end
```

Listing 11: With Macro

For Comprehension

```
alias :lists, as: Lists
alias :math, as: Math

for a <- Lists.seq(1,100),
    b <- Lists.seq(1,100),
    c <- Lists.seq(1,100),
    Math.pow(a,2) + Math.pow(b,2) == Math.pow(c, 2),
  do: [a: a, b: b, c: c]


[
  [a: 3, b: 4, c: 5],
  [a: 4, b: 3, c: 5],
  [a: 5, b: 12, c: 13],
  [a: 6, b: 8, c: 10],
  ...
]
```

Listing 8: For Comprehension

For Comprehension

In most languages this feature is called a list comprehension. The Elixir version can return other types.

```
for x <- [1,2,3], into: %{} do
  {x, x+10}
end

%{1 => 11, 2 => 12, 3 => 13}
```

Listing 9: For Comprehension

Using Macro

```elixir
defmodule FariWeb do
  def controller do
    quote do
      use Phoenix.Controller, namespace: FariWeb
      import Plug.Conn
      import FariWeb.Router.Helpers
      import FariWeb.Gettext
    end
  end

  defmacro __using__(which) when is_atom(which) do
    apply(__MODULE__, which, [])
  end
end

defmodule FariWeb.UserController do
  use FariWeb, :controller
end
```

Listing 10: Using Macro

Mix

Mix is a build tool that provides tasks for creating, compiling, and testing Elixir projects, managing its dependencies, and more.

- For those familiar with ruby mix is like:
  - bundler
  - rake
  - rails CLI

Plug

Plug is:

Plug

Plug is:

- Connection adapters for different web servers in the Erlang VM.

Plug

Plug is:

- Connection adapters for different web servers in the Erlang VM.
- A specification for composable modules between web applications.

Plug.Conn

The Plug.Conn is the main data structure. It is used to represent the Requests and create Responses.

Request fields

Name	Description
host	The requested host as a binary, example: "www.example.com"
path_info	The path split into segments, example: ["hello", "world"]
request_path	The requested path, example: /trailing/and//double//slashes/
port	The requested port as an integer, example: 80
remote_ip	The IP of the client, example: {151, 236, 219, 228}. This field is meant to be overwritten by plugs that understand e.g. the X-Forwarded-For header or HAProxys PROXY protocol. It defaults to peers IP
req_headers	The request headers as a list, example: ["content-type", "text/plain"]. Note all headers will be downcased
scheme	The request scheme as an atom, example: :http
query_string	The request query string as a binary, example: "foo=bar"

Fetchable fields

Name	Description
cookies	The request cookies with the response cookies
body_params	The request body params, populated through a Plug.Parsers parser.
query_params	The request query params, populated through fetch_query_params/2
path_params	The request path params, populated by routers such as Plug.Router
params	The request params, the result of merging the :body_params and :query_params with :path_params
req_cookies	The request cookies (without the response ones)

Response fields

Name	Description
resp_body	The response body, by default is an empty string. It is set to nil after the response is sent, except for test connections.
resp_charset	The response charset, defaults to utf-8
resp_cookies	The response cookies with their name and options
resp_headers	The response headers as a list of tuples, by default cache & control is set to "max-age=0, private, must-revalidate". Note, response headers are expected to have lower-case keys.
status	The response status

Connection fields

Name	Description
assigns	Shared user data as a map
owner	The Elixir process that owns the connection
halted	The boolean status on whether the pipeline was halted
secret_key_base	A secret key used to verify and encrypt cookies. the field must be set manually whenever one of those features are used. This data must be kept in the connection and never used directly, always use Plug.Crypto.KeyGenerator.generate/3 to derive keys from it
state	The connection state

Private fields

Name	Description
adapter	Holds the adapter information in a tuple
private	Shared library data as a map

Module Plug

```elixir
defmodule ContexthubWeb.Auth do
  use ContexthubWeb, :controller

  def init(opts), do: opts

  def call(conn, _opts) do
    ["Bearer " <> token] = get_req_header(conn, "authorization")
    case Guardian.decode_and_verify(token) do
      {:ok, _claims} -> assign(conn, :authenticated, true)
      {:error, _reason} ->
        conn
        |> put_status(401)
        |> json(%{error: "unauthenticated"})
        |> halt
    end
  end
end

defmodule ContexthubWeb.NotificationController do
  plug ContexthubWeb.Auth
end
```

Listing 12: Using Macro

Function Plug

```
defp resource(conn, _opt) do
  with {:ok, user} <- Fari.Auth.resource_from_token(conn) do
    conn
    |> Plug.Conn.put_private(:absinthe, %{context: %{current_user: user}})
  else
    {:error, :token_expired} ->
      conn
      |> Plug.Conn.put_private(:absinthe, %{context: %{token_expired: true}})

    error ->
      IO.inspect(error)
      conn
  end
end

plug(:resource)
```

Listing 13: Using Macro

Channels

Channels provide a means for bidirectional communication from clients that integrate with the Phoenix.PubSub layer for soft-realtime functionality.

- Phoenix.Transports.WebSocket
- Phoenix.Transports.LongPoll

Channels

You can write your own transports. Examples are:

- WebRTC
- Jabber
- RabbitMQ

What is GraphQL?

GraphQL is a data query language developed by Facebook in 2012 and release in 2015.

Definition

Query Language A language for the specification of procedures for the retrieval (and sometimes also modification) of information from a database.

Definition

Query Language A language for the specification of procedures for the retrieval (and sometimes also modification) of information from a database.

- The fact that GraphQL can be used as a web API is an implementation detail.

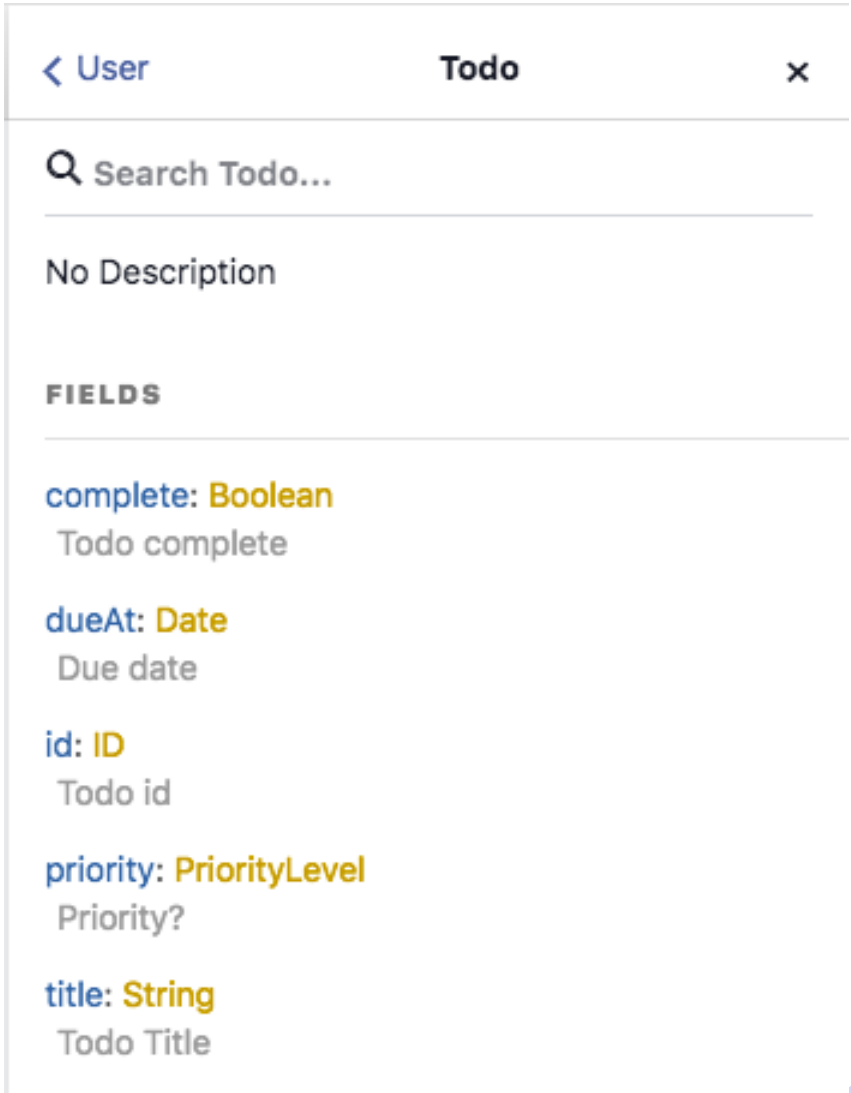
Documentation

Descriptions can be added to every field:

```
object :todo do
  field(:id, :id, description: "Todo id")
  field(:title, :string, description: "Todo Title")
  field(:priority, :priority_level, description: "Priority?")
  field(:due_at, :date, description: "Due date")
  field(:complete, :boolean, description: "Todo complete")
end
```

Listing 3: Object Type

Documentation - GraphiQL



Types

- APIs in GraphQL are organized by types and fields.

Builtin Types

- boolean
- float
- id
- integer
- string

Custom Types

- You can also create custom Scalar Types.

```
scalar :date do
  parse(fn input ->
    case Timex.parse!(input.value, "{YYYY}-{0M}-{D}") |> DateTime.from_naive("
        Etc/UTC") do
      {:ok, date} -> {:ok, date}
      _ -> :error
    end
  end)

  serialize(fn date -> Date.to_iso8601(date) end)
end
```

Listing 1: Custom Scaler Type

Enum Types

Enum types represents one of a finite set of possible values.

```
enum :priority_level, description: "Todo priority levels" do
  value :high, as: :true, description: "High Priority"
  value :low, as: :false, description: "Low Priority"
end
```

Listing 2: Enum Type

```
Request to pry #PID<0.623.0> at FariWeb.Resolvers.Todos.create/3 (lib/fari_web/resolvers/todos.ex:5)

    3:
    4:   def create(_obj, args, %{context: %{current_user: user}}) do
    5:     require IEx; IEx.pry
    6:     user
    7:     |> Ecto.build_assoc(:todos)

Allow? [Yn] y

Interactive Elixir (1.6.6) - press Ctrl+C to exit (type h() ENTER for help)
pry(1)> args
%{priority: true}
pry(2)> true == :true
true
```

Input Types

Input types defines a set of input fields; the input fields are either scalars, enums, or other input objects. This allows arguments to accept arbitrarily complex values.

```
@desc "Filtering options for the menu item list"
input_object :menu_item_filter do
  @desc "Matching a name"
  field :name, :string

  @desc "Matching a category name"
  field :category, :string

  @desc "Matching a tag"
  field :tag, :string

  @desc "Priced above a value"
  field :priced_above, :float

  @desc "Priced below a value"
  field :priced_below, :float
end
```

Listing 14: Input Types

Input Types

```
field :menu_items, list_of(:menu_item) do
  arg :filter, :menu_item_filter
  arg :order, type: :sort_order, default_value: :asc
  resolve &Resolvers.Menu.menu_items/3
end
```

Listing 15: Input Types

Object Types

Objects types represent a list of named fields, each of which yields a value of a specific type.

```
object :todo do
  field(:id, :id, description: "Todo id")
  field(:title, :string, description: "Todo Title")
  field(:priority, :priority_level, description: "Priority?")
  field(:due_at, :date, description: "Due date")
  field(:complete, :boolean, description: "Todo complete")
end
```

Listing 3: Object Type

Query

- The Query Root object is where all queries are found.

```
query do
  field :users, list_of(:user), description: "List users in my groups" do
    resolve(&FariWeb.Resolvers.Users.list_users/3)
  end
end
```

Listing 4: User Root Object Type

Resolvers

- The resolve function tells the object how to get it's data.

```
  def list_users(_obj, _args, %{context: %{current_user: user}}) do
    users =
      user
      |> Fari.Repo.preload(:groups)
      |> get_groups()
      |> Enum.map(fn group -> Fari.Repo.preload(group, :users) end)
      |> Enum.map(fn g -> g.users end)
      |> List.flatten()

    {:ok, users}
end
```

Listing 5: Resolve function

Resolver Arguments

- Resolve functions take 3 arguments.

Resolver Arguments

- Resolve functions take 3 arguments.
- parent
  - This is the object above the current object.
  - At the root level this is normally nil.

Resolver Arguments

- Resolve functions take 3 arguments.
- parent
  - This is the object above the current object.
  - At the root level this is normally nil.
- arguments

Resolver Arguments

- Resolve functions take 3 arguments.
- parent
  - This is the object above the current object.
  - At the root level this is normally nil.
- arguments
  - A map of the arguments

Resolver Arguments

- Resolve functions take 3 arguments.
- parent
  - This is the object above the current object.
  - At the root level this is normally nil.
- arguments
  - A map of the arguments
- Absinthe.Resolution Struct

Resolver Arguments

- Resolve functions take 3 arguments.
- parent
  - This is the object above the current object.
  - At the root level this is normally nil.
- arguments
  - A map of the arguments
- Absinthe.Resolution Struct
  - This contains the **context** and other execution data

Resolvers Return Value

- To have successful result, return a tuple in the form of `{:ok, value}`
- To have an error result, return a tuple in the form of `{:error, reason}`

Mutations

Mutations work the same way Queries do. The difference is side-effects are expected.

```
mutation do
  field :login, non_null(:session) do
    arg(:email, :string, description: "User's email")
    arg(:password, :string, description: "Password")

    resolve(&FariWeb.Resolvers.Users.login/3)
  end
end
```

Listing 22: Mutations

Subscriptions

Subscriptions allow users to request data updates.

```
subscription do
  field :marked_todo, :todo do
    config(fn _args, _info ->
      {:ok, topic: "*"}
    end)

    resolve(fn todo, _, _ ->
      {:ok, todo}
    end)
  end
end

defp notify_mark({:ok, todo}) do
  Absinthe.Subscription.publish(FariWeb.Endpoint, todo, marked_todo: "*")
  {:ok, todo}
end
```

Listing 16: Subscriptions

Authentication

```
defp resource(conn, _opt) do
  with {:ok, user} <- Fari.Auth.resource_from_token(conn) do
    conn
    |> Plug.Conn.put_private(:absinthe, %{context: %{current_user: user}})
  else
    {:error, :token_expired} ->
      conn
      |> Plug.Conn.put_private(:absinthe, %{context: %{token_expired: true}})

    error ->
      IO.inspect(error)
      conn
  end
end

plug(:resource)
```

Listing 13: Using Macro

Authentication

```elixir
defmodule Fari.Authentication do
  @behaviour Absinthe.Middleware
  @moduledoc false

  def call(resolution, config) do
    case resolution.context do
      %{current_user: _} ->
        resolution

      %{token_expired: true} ->
        resolution
        |> Absinthe.Resolution.put_result({:error, "token expired"})

      _ ->
        resolution
        |> Absinthe.Resolution.put_result({:error, "unauthenticated"})
    end
  end
end
```

Listing 18: Authentication

Authentication

```
  def middleware(middleware, %Field{identifier: id}, _object) when id in [:
      register, :login] do
    middleware
  end

  def middleware(middleware, _field, %Object{identifier: id}) when id in [:query
      , :mutation] do
    [Fari.Authentication | middleware]
  end

  def middleware(middleware, _field, _object) do
    middleware
  end
}
```

Listing 17: Authentication

Data Loader

```
def dataloader() do
  Dataloader.new()
  |> Dataloader.add_source(Fari.Core.User, Fari.Core.data())
  |> Dataloader.add_source(Fari.Core.Todo, Fari.Core.data())
end

def context(ctx) do
  Map.put(ctx, :loader, dataloader())
end

def plugins do
  [Absinthe.Middleware.Dataloader | Absinthe.Plugin.defaults()]
end
```

Listing 19: Data Loader

Data Loader

```elixir
defmodule Fari.Core do
  alias Fari.Repo
  import Ecto.Query

  def data() do
    Dataloader.Ecto.new(Repo, query: &query/2)
  end

  # def query(Fari.Core.Todo, _args) do
  #   from(t in Fari.Core.Todo, where: t.complete == ^false)
  # end

  def query(queryable, _args) do
    queryable
  end
end
```

Listing 20: Data Loader

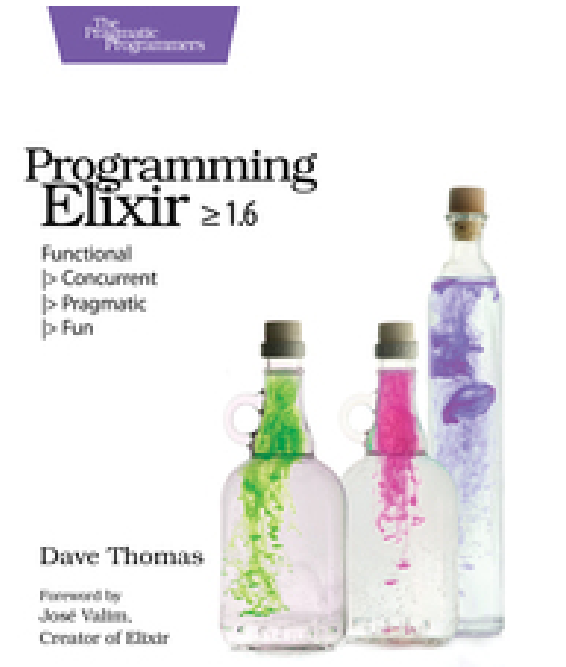
Data Loader

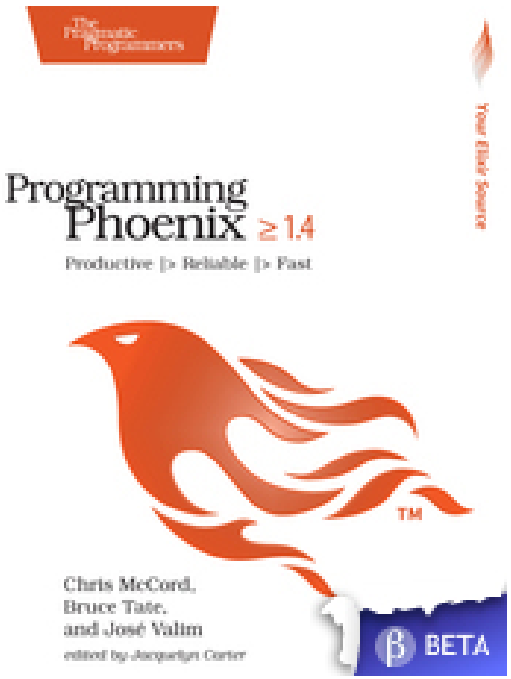
```
object :user do
  field(:id, :id, description: "User ID")
  field(:email, :string, description: "Email")
  field(:first_name, :string, description: "First Name")
  field(:last_name, :string, description: "Last Name")

  field :full_name, :string, description: "Full name" do
    resolve(fn user, _, _ ->
      {:ok, "#{user.first_name} #{user.last_name}"}
    end)
  end

  field(
    :groups,
    list_of(:group),
    description: "User Groups",
    resolve: dataloader(Fari.Core.User, :groups)
  )

  field(
    :todos,
    list_of(:todo),
    description: "User todos",
    resolve: dataloader(Fari.Core.Todo, :todos)
  )
end
```

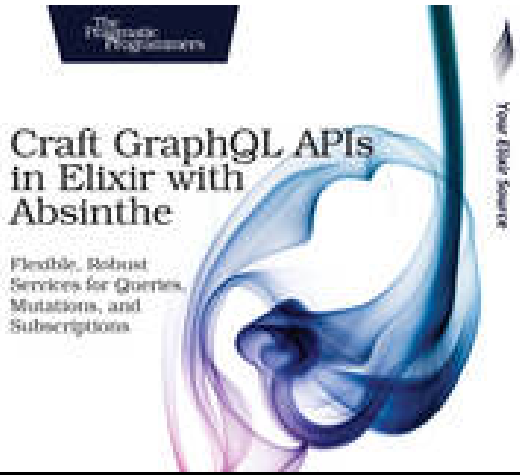
Listing 21: Data Loader

Questions

Thank you


The Pragmatic Programmers
Programming Elixir ≥ 1.6
Functional
|> Concurrent
|> Pragmatic
|> Fun
Dave Thomas
Foreword by
José Valim,
Creator of Elixir


The Pragmatic Programmers
Your Elixir Source
Programming Phoenix ≥ 1.4
Productive |> Reliable |> Fast
Chris McCord,
Bruce Tate,
and José Valim
edited by Jacquelyn Carter
BETA


The Pragmatic Programmers
Your Elixir Source
Craft GraphQL APIs in Elixir with Absinthe
Flexible, Robust Services for Queries, Mutations, and Subscriptions