

Phoenix Is Not Your Application

Lance Halvorsen
ElixirConf EU 2016

Phoenix is one
of your applications.

Our Language Gives
Us Away

We Say

I'm building a Phoenix app.

Or a Rails app.

Or an Ember app.

Or a React app.

Is That True?

- No!
- Well, currently it might be.
- But, it doesn't have to be.

Ideally We Would Say

I'm building a logistics app.

Or an analytics app.

Or a physics engine.

Or a fortune cookie app.

. . . with a web interface.

Your application has its
own domain.

Phoenix is the interface.

And Phoenix has its own domain as well.

A Thought Experiment

We have a large application written in Rails.

It has, say, 100 models and 75 controllers.

It's been migrated from Rails 2 to Rails 3 to Rails 4.

The tests are "pretty good".

We now need to port this application to Rhoda, or Lotus, or . . .

How does that feel?

(My palms are sweating.)

Why does this seem so
daunting?

Our application logic is
entangled with the
interface.

Our Application Logic is Dispersed

- across models
- across controllers
- across views
- maybe even across templates

Maintainability Becomes an Issue

- More difficult to isolate and extract pieces as the application grows.
- More difficult to localize problems when the domains are mixed.

Portability Becomes an Issue

- Changing frameworks is almost certainly going to require a rewrite.
- Since the logic is dispersed, it's easy to miss functionality during the rewrite.
- Tests will help, but even excellent coverage can't guarantee that you'll catch everything.

The solution is to build our application separately first, and layer the interface on top later.

How does this happen?

It begins with the HTTP
request/response cycle.

HTTP is stateless.

But applications are not.

We store state in the database.

We need to get state on each request.

We need to write state back to the db after change.

We lean on the framework for data access.

We model our domain entities for databases.

Consider the Difference

```
class Comment
```

```
class Comment < ActiveRecord::Base
```

```
class Comment
  # do comment things
end
```

```
class Comment < ActiveRecord::Base
  # do comment things
  # handle database connections
  # model database relationships
  # validate data
  # generate and execute SQL
  # and the list goes on
end
```


A Comment is part of our domain.

A Comment model is part of ActiveRecord's domain.

Ecto Has a Better Story

- There are no models, only schemas.
- There is good separation of concerns.
- But we're still thinking in database terms.

Models are not part of
our domain.

A router probably isn't
part of our domain either.

Or a controller.

Or a view.

We need to decouple our domain from the framework's domain.

Both in the code and
in our heads.

OTP has an answer.

Applications

An OTP Application is a Behaviour.

A Behaviour Is

- A distillation of years of experience by the Erlang team
- A Design pattern
- A module of code common to each Behaviour (the wiring and plumbing)
- A list of callbacks for customization


But what is an Application, really?

- It's an independent module or group of modules.
- It implements a specific piece of functionality.
- It can be started and stopped independently.
- It is supervised.
- It can be reused in other OTP Applications.

We already use them all the time.

Every mix.exs file will have a list of dependent OTP Applications.

```
def application do
  [mod: {HelloPhoenix, []},
   applications[:phoenix, :phoenix_html, :cowboy,
                :logger, :gettext, :phoenix_ecto,
                :postgrex]]
end
```



We can see our Applications
with `:observer.start`.

Let's say we have a freshly generated
Phoenix application called
hello_phoenix.


```
$ iex -S mix phoenix.server
```

```
iex> :observer.start
```

System

Load Charts

Memory Allocators

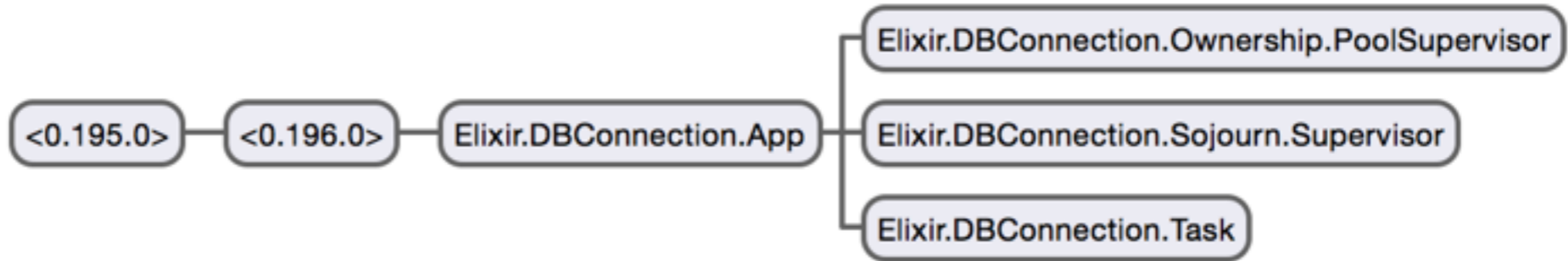
Applications

Processes


Table Viewer

Trace Overview

- cowboy
- db_connection
- ecto
- elixir
- hello_phoenix
- hex
- iex
- inets
- kernel
- logger
- mix
- phoenix
- plug
- postgrex
- ranch
- ssl



For our `hello_phoenix` app, we define the `Application` module in `lib/hello_phoenix.ex`.

```
defmodule HelloPhoenix do
  use Application 
  def start(_type, _args) do
    import Supervisor.Spec, warn: false
    children = [
      supervisor(HelloPhoenix.Endpoint, []),
      supervisor(HelloPhoenix.Repo, [])
    ]
    opts = [strategy: :one_for_one, name: HelloPhoenix.Supervisor]
    Supervisor.start_link(children, opts)
  end
  def config_change(changed, _new, removed) do
    HelloPhoenix.Endpoint.config_change(changed, removed)
  end
end
```


```
defmodule HelloPhoenix do
  use Application
  def start(_type, _args) do ←
    import Supervisor.Spec, warn: false
    children = [
      supervisor(HelloPhoenix.Endpoint, []),
      supervisor(HelloPhoenix.Repo, []),
    ]
    opts = [strategy: :one_for_one, name: HelloPhoenix.Supervisor]
    Supervisor.start_link(children, opts)
  end
  def config_change(changed, _new, removed) do
    HelloPhoenix.Endpoint.config_change(changed, removed)
  end
end
```

```
defmodule HelloPhoenix do
  use Application
  def start(_type, _args) do
    import Supervisor.Spec, warn: false
    children = [
      supervisor(HelloPhoenix.Endpoint, []),
      supervisor(HelloPhoenix.Repo, []),
    ]
    opts = [strategy: :one_for_one, name: HelloPhoenix.Supervisor]
    Supervisor.start_link(children, opts)
  end
  def config_change(changed, _new, removed) do
    HelloPhoenix.Endpoint.config_change(changed, removed)
  end
end
```

```
defmodule HelloPhoenix do
  use Application
  def start(_type, _args) do
    import Supervisor.Spec, warn: false
    children = [
      supervisor(HelloPhoenix.Endpoint, []),
      supervisor(HelloPhoenix.Repo, [])
    ]
    opts = [strategy: :one_for_one, name: HelloPhoenix.Supervisor]
    Supervisor.start_link(children, opts)
  end
  def config_change(changed, _new, removed) do
    HelloPhoenix.Endpoint.config_change(changed, removed)
  end
end
```

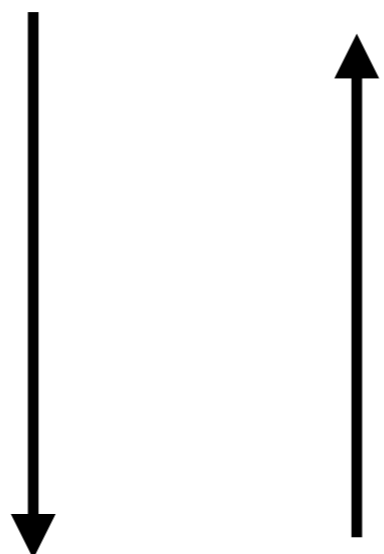


```
defmodule HelloPhoenix do
  use Application
  def start(_type, _args) do
    import Supervisor.Spec, warn: false
    children = [
      supervisor(HelloPhoenix.Endpoint, []),
      supervisor(HelloPhoenix.Repo, [])
    ]
    opts = [strategy: :one_for_one, name: HelloPhoenix.Supervisor]
    Supervisor.start_link(children, opts)
  end
  def config_change(changed, _new, removed) do
    HelloPhoenix.Endpoint.config_change(changed, removed)
  end
end
```

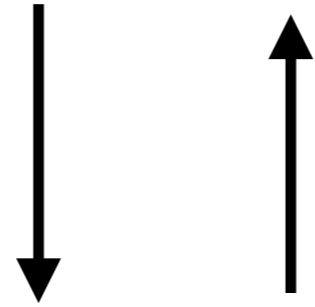


How does this help us
separate domains?

- We can build our application in pure Elixir.
- And generate a new Phoenix application.
- Then import our Elixir application into our Phoenix application.
- Finally, we can have the Phoenix domain elements talk to our real domain, in the Elixir application.



Big
Box
App



Interface



Application

Let's Try It Out



Introduction to Mix

- 1 [Our first project](#)
- 2 [Project compilation](#)
- 3 [Running tests](#)
- 4 [Environments](#)
- 5 [Exploring](#)

In this guide, we will learn how to build a complete Elixir application, with its own supervision tree, configuration, tests and more.

The application works as a distributed key-value store. We are going to organize key-value pairs into buckets and distribute those buckets across multiple nodes. We will also build a simple client that allows us to connect to any of those nodes and send requests such as:

```
CREATE shopping
OK

PUT shopping milk 1
OK

PUT shopping eggs 3
OK

GET shopping milk
1
OK
```

News: [Elixir v1.2 released](#)

GETTING STARTED

1. [Introduction](#)
2. [Basic types](#)
3. [Basic operators](#)
4. [Pattern matching](#)
5. [case, cond and if](#)
6. [Binaries, strings and char lists](#)
7. [Keywords and maps](#)
8. [Modules](#)
9. [Recursion](#)
10. [Enumerables and streams](#)
11. [Processes](#)
12. [IO and the file system](#)
13. [alias, require and import](#)
14. [Module attributes](#)
15. [Structs](#)
16. [Protocols](#)
17. [Comprehensions](#)
18. [Sigils](#)

Step 1

Generate a new mix application

```
$ mix new kv --module KV --sup
```



```
* creating README.md
```

```
* creating .gitignore
```

```
* creating mix.exs
```

```
* creating config
```

```
* creating config/config.exs
```

```
* creating lib
```

```
* creating lib/kv.ex
```

```
* creating test
```

```
* creating test/test_helper.exs
```

```
* creating test/kv_test.exs
```

Your Mix project was created successfully.

You can use "mix" to compile it, test it, and more:

```
cd kv
```

```
mix test
```

Run "mix help" for more commands.

Step 1.5

Follow the rest of the tutorial :^)

Step 2.

Build out the KV GenServer

```
defmodule KV.Registry do
  use GenServer

  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end

  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end

  def create(server, name) do
    GenServer.call(server, {:create, name})
  end

  def init(:ok) do
    {:ok, %{}}
  end

  def handle_call({:create, name}, _from, names) do
    if Map.has_key?(names, name) do
      {:reply, name, names}
    else
      {:ok, bucket} = KV.Bucket.start_link
      names = Map.put(names, name, bucket)
      {:reply, name, names}
    end
  end
end
```

```
defmodule KV.Registry do
  use GenServer

  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end

  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end

  def create(server, name) do
    GenServer.call(server, {:create, name})
  end

  # callbacks go here
end
```

```
defmodule KV.Registry do
  # client functions go here

  def init(:ok) do
    {:ok, %{}}
  end

  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end

  def handle_call({:create, name}, _from, names) do
    if Map.has_key?(names, name) do
      {:reply, name, names}
    else
      {:ok, bucket} = KV.Bucket.start_link
      names = Map.put(names, name, bucket)
      {:reply, name, names}
    end
  end
end
```


```
defmodule KV.Registry do
  use GenServer
end
```

```
defmodule KV.Registry do
  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end


  ...

  def init(:ok) do
    {:ok, %{}}
  end
end
```

```
defmodule KV.Registry do
  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end
  ...
  def init(:ok) do
    {:ok, %{}}
  end
end
```




```
defmodule KV.Registry do
  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end
  ...
  def init(:ok) do
    {:ok, %{}}
  end
end
```



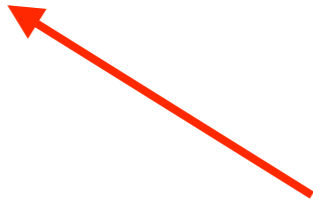
```
defmodule KV.Registry do
  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end

  ...

  def init(:ok) do
    {:ok, %{}}
```

```
defmodule KV.Registry do
  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end
  ...

  def init(:ok) do
    {:ok, %{}}
  end
end
```



```
defmodule KV.Registry do
  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end

  ...

  def init(:ok) do
    {:ok, %{}}
  end
end
```

```
defmodule KV.Registry do
  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end

  ...

  def init(:ok) do
    { :ok, %{} }
  end
end
```

```
defmodule KV.Registry do
  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: :kv_registry)
  end

  ...

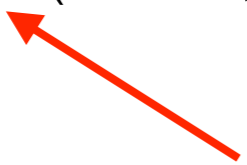
  def init(:ok) do
    {:ok, %{}}
  end
end
```

```
defmodule KV.Registry do
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end
end
```

```
defmodule KV.Registry do
  :kv_registry
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end
end
```



```
defmodule KV.Registry do
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end
end
```



```
defmodule KV.Registry do
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end
end
```

```
defmodule KV.Registry do
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end
end
```

```
defmodule KV.Registry do
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end
end
```

```
defmodule KV.Registry do
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end
end
```

```
defmodule KV.Registry do
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names} ←
  end
end
```

```
defmodule KV.Registry do
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end
end
```

```
defmodule KV.Registry do
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end
  ...
  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end
end
```



```
defmodule KV.Registry do
  def create(server, name) do
    GenServer.call(server, {:create, name})
  end
  ...
  def handle_call({:create, name}, _from, names) do
    if Map.has_key?(names, name) do
      {:reply, name, names}
    else
      {:ok, bucket} = KV.Bucket.start_link
      names = Map.put(names, name, bucket)
      {:reply, name, names}
    end
  end
end
end
```

```
defmodule KV.Registry do
  def create(server, name) do
    GenServer.call(server, {:create, name})
  end
  ...
  def handle_call({:create, name}, _from, names) do
    if Map.has_key?(names, name) do
      {:reply, name, names}
    else
      {:ok, bucket} = KV.Bucket.start_link
      names = Map.put(names, name, bucket)
      {:reply, name, names}
    end
  end
end
end
```

```
defmodule KV.Registry do
  def create(server, name) do
    GenServer.call(server, {:create, name})
  end
  ...
  def handle_call({:create, name}, _from, names) do
    if Map.has_key?(names, name) do
      {:reply, name, names}
    else
      {:ok, bucket} = KV.Bucket.start_link
      names = Map.put(names, name, bucket)
      {:reply, name, names}
    end
  end
end
end
```

Step 4.

Register the GenServer as a Worker

- Add the KV.Registry module as a worker in lib/kv.ex.
- This ensures the worker will start when the application does.

```
defmodule KV do
  use Application

  def start(_type, _args) do
    import Supervisor.Spec, warn: false

    children = [
      worker(KV.Registry, []) ←
    ]

    opts = [strategy: :one_for_one, name: KV.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Step 5.

Generate a new Phoenix application

- Call it kv_interface
- Generate it without Ecto

```
$ mix phoenix.new kv_interface --no-ecto ←
```

```
* creating kv_interface/config/config.exs  
* creating kv_interface/config/dev.exs
```

```
<snip>
```

```
* creating kv_interface/web/views/layout_view.ex  
* creating kv_interface/web/views/page_view.ex
```

```
Fetch and install dependencies? [Yn] y
```

```
* running mix deps.get  
* running npm install && node node_modules/brunch/bin/brunch build
```

We are all set! Run your Phoenix application:

```
$ cd kv_interface  
$ mix phoenix.server
```

You can also run your app inside IEx (Interactive Elixir) as:

```
$ iex -S mix phoenix.server
```

Step 6.

Bring KV into kv_interface

- Add it as a dependency in mix.exs.

```
defp deps do
```

```
  [{:phoenix, "~> 1.1.4"},
```

```
   {:phoenix_html, "~> 2.4"},
```

```
   {:phoenix_live_reload, "~> 1.0", only: :dev},
```

```
   {:gettext, "~> 0.9"},
```

```
   {:cowboy, "~> 1.0"},
```

```
   {:kv, path: "../kv"}]
```



```
end
```

Step 7.

Add KV to the application list

- In `mix.exs`.

```
def application do
```

```
  [mod: {KvInterface, []},
```

```
   applications: [:phoenix, :phoenix_html, :cowboy,
```

```
                  :logger, :gettext, :kv]]
```

```
end
```



Re-check
with `:observer.start`

System

Load Charts

Memory Allocators

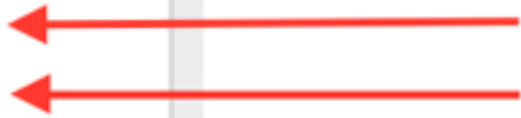
Applications

Processes

Table Viewer

Trace Overview

- cowboy
- elixir
- hex
- iex
- inets
- kernel
- kv
- kv_interface
- logger
- mix
- phoenix
- plug
- ranch
- ssl



- <0.444.0>
- Elixir.KvInterface.Endpoi
- Elixir.KvInterface.Endpoi

Get Them Talking Together

- Add a controller at `web/controllers/registry_controller.ex`.

```
defmodule KvInterface.RegistryController do
  use KvInterface.Web, :controller
end
```

Add a show Action

```
def show(conn, %{"name" => name}) do
  item = KV.Registry.lookup(:kv_registry, name)
  render(conn, "show.html", item: item)
end
```



Add a create Action

```
def create(conn, %{"name" => name}) do
  item = KV.Registry.create(:kv_registry, name)
  render(conn, "show.html", item: item)
end
```


What have we gained?

- Isolation!
 - Develop in isolation
 - Test in isolation
- Domain separation
 - Maintainability
 - Portability



Productive. Reliable. Fast.

A productive web framework that does not compromise speed and maintainability.

Build APIs, HTML 5 apps & more

[See our guides](#)

HOW IS PHOENIX DIFFERENT?

Phoenix brings back the simplicity and joy in writing modern web applications by mixing tried and true technologies with a fresh breeze of functional ideas.

[Get started with Phoenix](#)

BUILDING THE NEW WEB

Create rich, interactive experiences across browsers, native mobile apps, and embedded devices with our real-time streaming technology called Channels.

[Learn about channels](#)

BATTLE-PROVEN TECHNOLOGY

Phoenix leverages the Erlang VM ability to handle millions of connections alongside Elixir's beautiful syntax and productive tooling for building fault-tolerant systems.

[More about Elixir & the Erlang VM](#)

Thank you!
That's a wrap.

And this is me:
[@lance_halvorsen](#)
[github/lancehalvorsen](https://github.com/lancehalvorsen)