

Elixir & Phoenix 101

¿QUIÉN SOY?

DESARROLADORA JUNIOR FRONTEND

GRADUADA EN INGENIERÍA INFORMÁTICA UDC

BEAMER DESDE ENERO DE 2022

CURIOSA Y EN CONSTANTE APRENDIZAJE



CARLA RODRÍGUEZ ESTÉVEZ

DE QUÉ VENGO A HABLAR

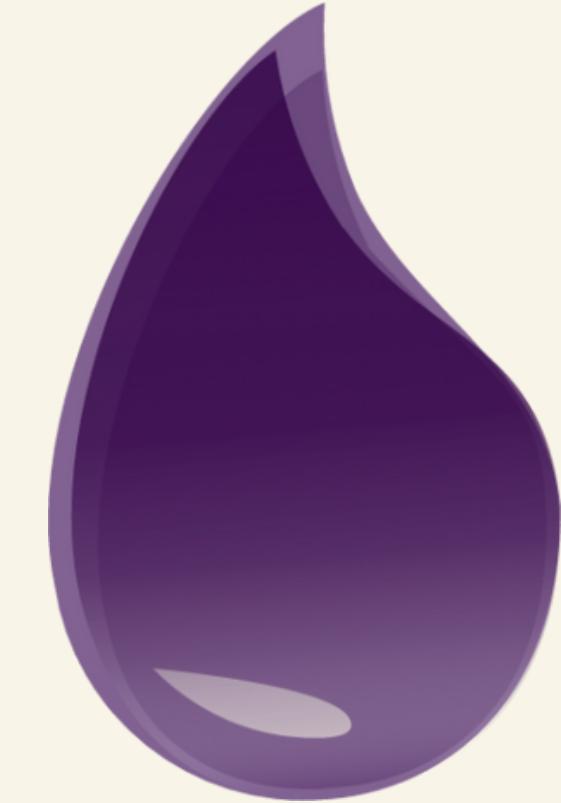
Elixir

- Lenguaje de programación funcional
- Escalable y tolerante a fallos.
- Creado para manejar concurrencia y paralelismo.
- Permite crear sistemas distribuidos.
- Se ejecuta en la máquina virtual de Erlang.

Phoenix

- Framework web de Elixir.
- Hereda los puntos positivos del lenguaje.
- Buen rendimiento de las aplicaciones.
- Tiene una amplia variedad de librerías para funcionalidad.

POR QUÉ ES IMPORTANTE SABERLO

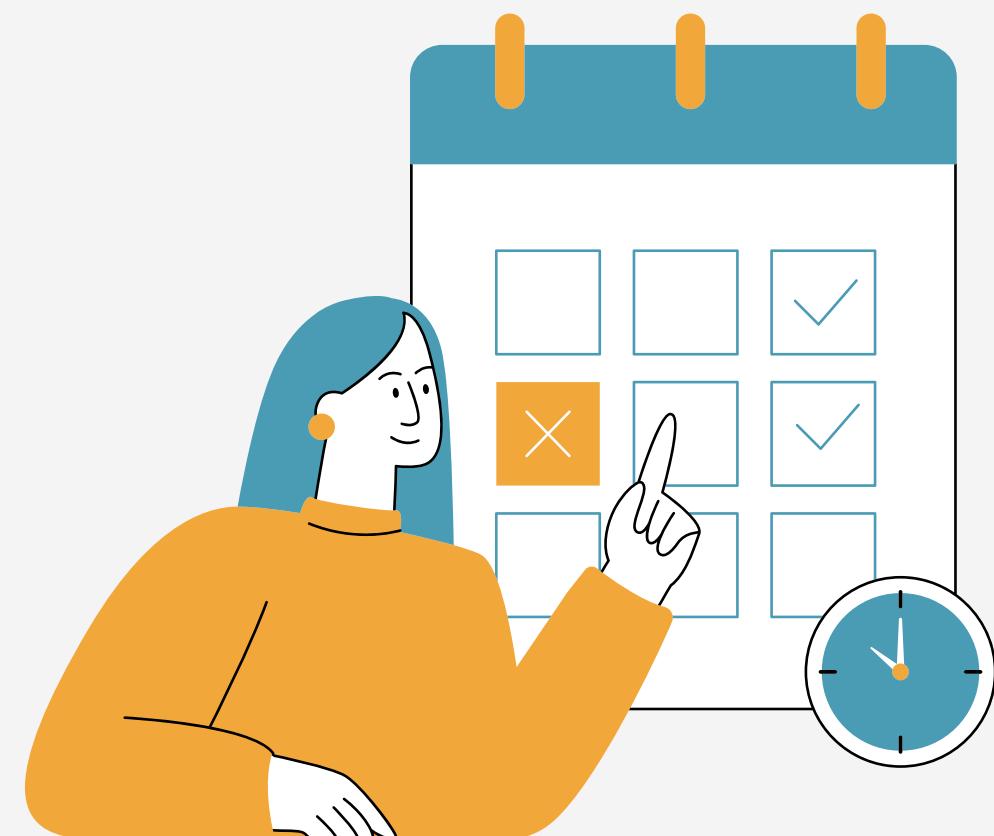


¿EN QUÉ LO USAN?

- Tecnologías y código en el backend.
- Sistemas de mensajería en tiempo real.
- Llamadas grupales.
- Servicios con arquitecturas p2p.
- ...

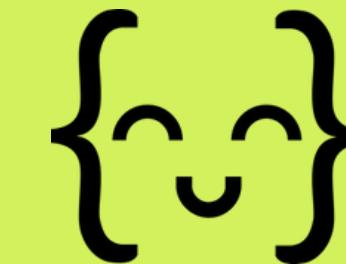


Básicos de Elixir



Plataformas recomendadas:

- Exercism



- Elixir School



- Udemy

- Comunidades Elixir

TIPOS BÁSICOS

Enteros

```
iex> 255  
255
```

Booleanos

```
iex> true  
true  
iex> false  
false
```

Strings

```
iex> name = "Sean"  
"Sean"  
iex> "Hello " <>> name  
"Hello Sean"
```

Flotantes

```
iex> 3.14  
3.14  
iex> .14  
** (SyntaxError) iex:2: syntax error before: '.'  
iex> 1.0e-10  
1.0e-10
```

Átomos

```
iex> :foo  
:foo  
iex> :foo == :bar  
false
```

OPERACIONES BÁSICAS

Aritméticas

```
iex> 2 + 2  
4  
iex> 2 - 1  
1  
iex> 2 * 5  
10  
iex> 10 / 5  
2.0
```

Booleanas

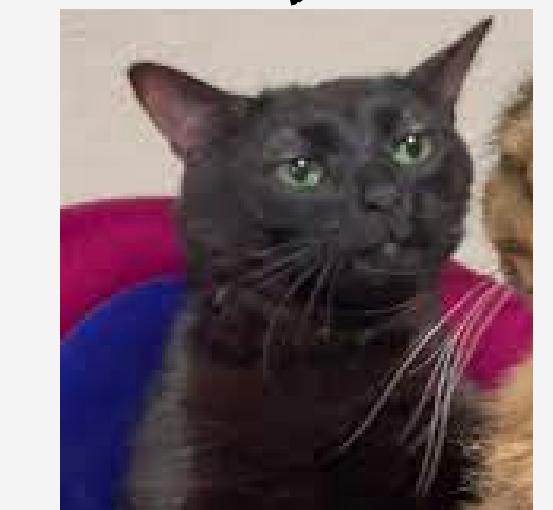
```
iex> -20 || true  
-20  
iex> false || 42  
42  
  
iex> 42 && true  
true  
iex> 42 && nil  
nil  
  
iex> !42  
false  
iex> !false  
true
```

Comparaciones

```
iex> 2 == 2.0  
true  
iex> 2 === 2.0  
false
```

```
iex> :hello > 999  
true  
iex> {:hello, :world} > [1, 2, 3]  
false
```

Reacción normal



COLECCIONES MÁS COMUNES

Listas

```
iex> list = [3.14, :pie, "Apple"]
[3.14, :pie, "Apple"]
# Prepending (fast)
iex> ["π" | list]
["π", 3.14, :pie, "Apple"]
# Appending (slow)
iex> list ++ ["Cherry"]
[3.14, :pie, "Apple", "Cherry"]
```

```
iex> [head | tail] = [3.14, :pie, "Apple"]
[3.14, :pie, "Apple"]
iex> head
3.14
iex> tail
[:pie, "Apple"]
```

Mapas

```
iex> map = %{:foo => "bar", "hello" => :world}
%{:foo => "bar", "hello" => :world}
iex> map[:foo]
"bar"
iex> map["hello"]
:world
```

```
iex> %{foo: "bar", hello: "world"}
%{foo: "bar", hello: "world"}
iex> %{foo: "bar", hello: "world"} == %{:foo => "bar", :hello => "world"}
true
```

```
iex> map = %{foo: "bar", hello: "world"}
%{foo: "bar", hello: "world"}
iex> map.hello
"world"
```

Tuplas

```
iex> {3.14, :pie, "Apple"}
{3.14, :pie, "Apple"}
```

PATTERN MATCHING

- El operador = no asigna un valor.
- Busca unos valores para que la expresión de la izquierda sea igual que la expresión de la izquierda.

```
# Tuples
iex> {:ok, value} = {:ok, "Successful!"}
{:ok, "Successful!"}
iex> value
"Successful!"
iex> {:ok, value} = {:error}
** (MatchError) no match of right hand side value: {:error}
```

Con funciones

```
iex> handle_result = fn
...>   {:ok, result} -> IO.puts "Handling result..."
...>   {:ok, _} -> IO.puts "This would be never run as"
...>   {:error} -> IO.puts "An error has occurred!"
...> end
```

```
iex> some_result = 1
1
iex> handle_result.({:ok, some_result})
Handling result...
:ok
iex> handle_result.({:error})
An error has occurred!
:ok
```

FUNCIONES Y MÓDULOS

- Pueden ser anónimas.

```
iex> sum = fn (a, b) -> a + b end  
iex> sum.(2, 3)  
5
```

- Se organizan en módulos
- Poseen nombre y aridad

```
defmodule Greeter2 do  
  def hello(), do: "Hello, anonymous person!" # hello/0  
  def hello(name), do: "Hello, " <> name # hello/1  
  def hello(name1, name2), do: "Hello, #{name1} and #{name2}"  
                                # hello/2  
end  
  
iex> Greeter2.hello()  
"Hello, anonymous person!"  
iex> Greeter2.hello("Fred")  
"Hello, Fred"  
iex> Greeter2.hello("Fred", "Jane")  
"Hello, Fred and Jane"
```

También hay funciones privadas, guardas, valores por defecto...

```
defmodule Greeter do  
  def hello(names, language_code \\ "en")  
  
  def hello(names, language_code) when is_list(names) do  
    names = Enum.join(names, ", ")  
  
    hello(names, language_code)  
  end  
  
  def hello(name, language_code) when is_binary(name) do  
    phrase(language_code) <> name  
  end  
  
  defp phrase("en"), do: "Hello, "  
  defp phrase("es"), do: "Hola, "  
end  
  
iex> Greeter.hello ["Sean", "Steve"]  
"Hello, Sean, Steve"  
  
iex> Greeter.hello ["Sean", "Steve"], "es"  
"Hola, Sean, Steve"
```

MIX

- Herramienta fundamental de Elixir.
- Nos ayuda a distintas tareas en proyectos de mayores dimensiones.
- Nos permite usar unos comandos básicos para creación de proyectos, generación de código, compilación y ejecutar acciones (testing, build...).
- Maneja las dependencias del proyecto.
- Permite trabajar con entornos.

```
def deps do
  [
    {:phoenix, "~> 1.1 or ~> 1.2"},
    {:phoenix_html, "~> 2.3"},
    {:cowboy, "~> 1.0", only: [:dev, :test]},
    {:slime, "~> 0.14"}
  ]
end
```

```
mix new example
```

```
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/example.ex
* creating test
* creating test/test_helper.exs
* creating test/example_test.exs
```

TIPOS Y ESPECIFICACIONES

- Sirve para definir interfaces de funciones.
- La documentación es útil, pero no se comprueba en tiempo de compilación.
- Elixir es un lenguaje dinámico. El compilador ignorará la información de tipos.
- Hay herramienta de QA que usan esta notación (Dialyzer).

```
defmodule Examples do
  defstruct first: nil, last: nil

  @type t(first, last) :: %Examples{first: first, last: last}

  @type t :: %Examples{first: integer, last: integer}
end
```

```
@spec sum_times(integer, Examples.t()) :: integer
def sum_times(a, params) do
  for i <- params.first..params.last do
    i
  end
  |> Enum.map(fn el -> el * a end)
  |> Enum.sum()
  |> round
end
```

BEHAVIOURS

- Para los amantes de Java...
- Funcionalidad casi idéntica a las interfaces

```
defmodule Parser do
  @doc """
  Parses a string.

  """

  @callback parse(String.t) :: {:ok, term} | {:error, atom}

  @doc """
  Lists all supported file extensions.

  """

  @callback extensions() :: [String.t]
end
```

```
defmodule JSONParser do
  @behaviour Parser

  @impl Parser
  def parse(str), do: {:ok, "some json " <> str} # ... parse JSON
```

```
@impl Parser
def extensions, do: [".json"]
end
```

```
defmodule CSVParser do
  @behaviour Parser

  @impl Parser
  def parse(str), do: {:ok, "some csv " <> str} # ... parse CSV

  @impl Parser
  def extensions, do: [".csv"]
end
```

PROCESOS

- Elixir y Erlang son languages concurrentes.
- Se pueden crear aplicaciones sin manejarlos directamente (lo que haremos).
- TODO es un proceso en Elixir.
- Los procesos tienen un mailbox con los mensajes recibidos.
- Los procesos solo leen un mensaje a la vez.
- Se pueden nombrar procesos, comprobar que un proceso existe, manejar timeouts...

```
pid = spawn fn ->
  receive do
    msg -> IO.puts("received: #{msg}")
  end
end
send(pid, "hello")
```

Spawn -> crea un proceso y devuelve su PID
Send -> envía un mensaje al PID indicado
Receive -> bloquea el proceso hasta recibir un mensaje

Más avanzado: supervisores, link processes, GenServers...

Phoenix

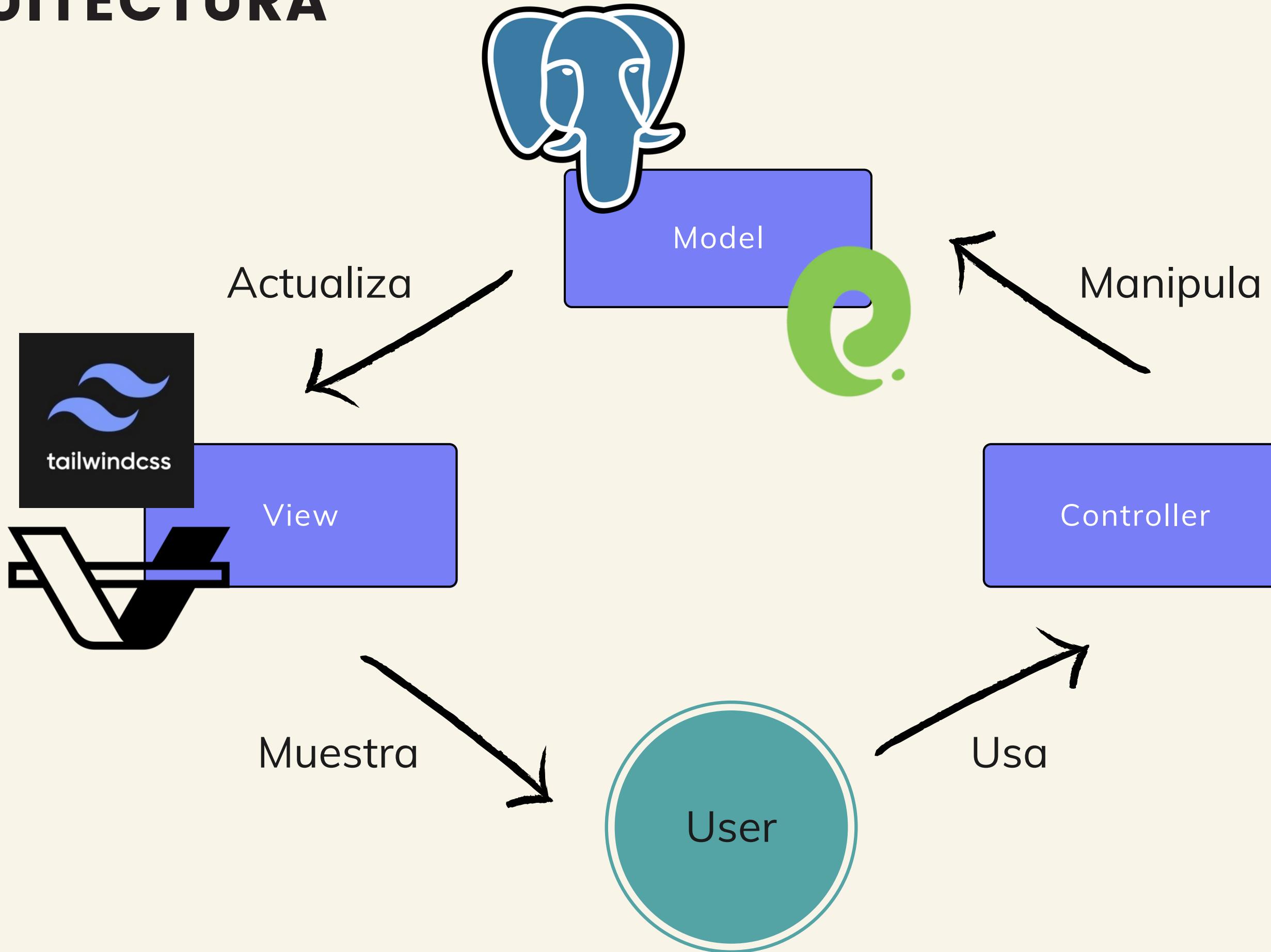


¿QUÉ ES?

- Framework para desarrollo de aplicaciones web en Elixir.
- Patrón Model-View-Controller.
- Alta productividad y buen rendimiento de la aplicación.
- Permite aplicaciones en tiempo real.
- Gran número de herramientas y librerías para aprovechar el framework.
- Compatibilidad con varios tipos de bases de datos.
- Permite un desarrollo de software completo.



ARQUITECTURA



Estructura por defecto de un proyecto de phoenix

Demo & tutorial



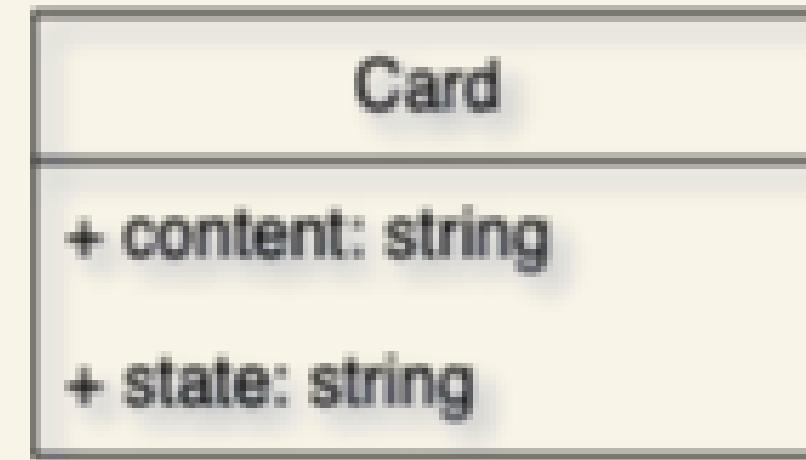
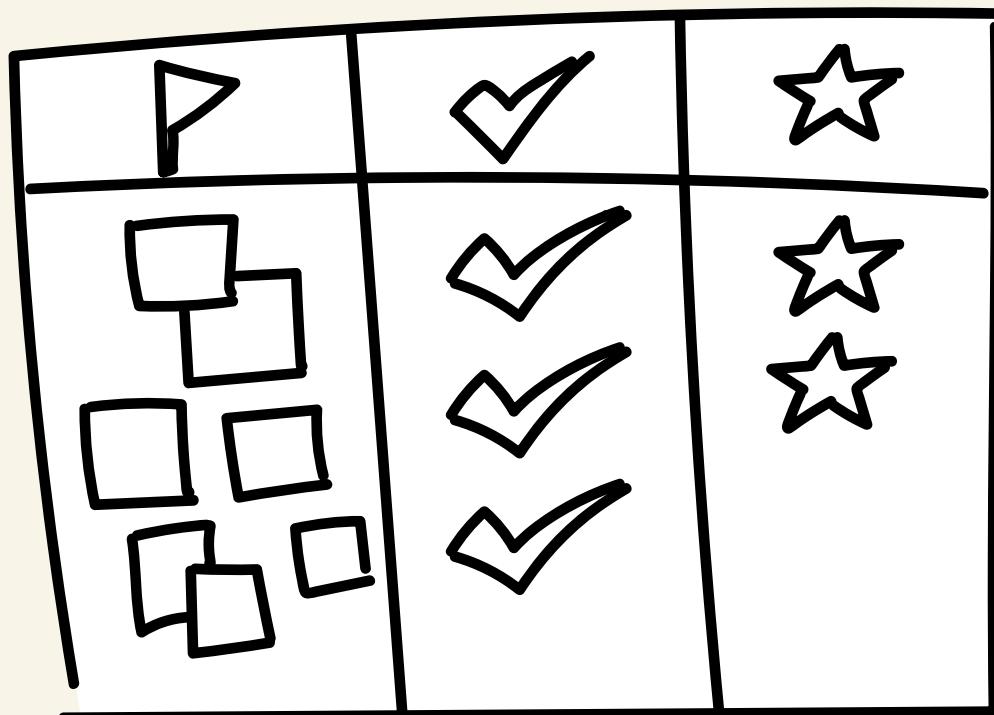
Prerrequisitos:

- Elixir y Phoenix instalado.
- PostgreSQL instalado y funcionando.
- Versiones:
 - Elixir: ≥ 1.12
 - Phoenix: ≥ 1.6
- Erlang (opcional)

IDEA BASE

- Modelo de tablero Kanban.
- Tarjetas con el contenido básico: Estado de la tarjeta y contenido de la tarjeta.
- Funcionamiento en tiempo real.

The possible values of state will be Todo, In progress and completed

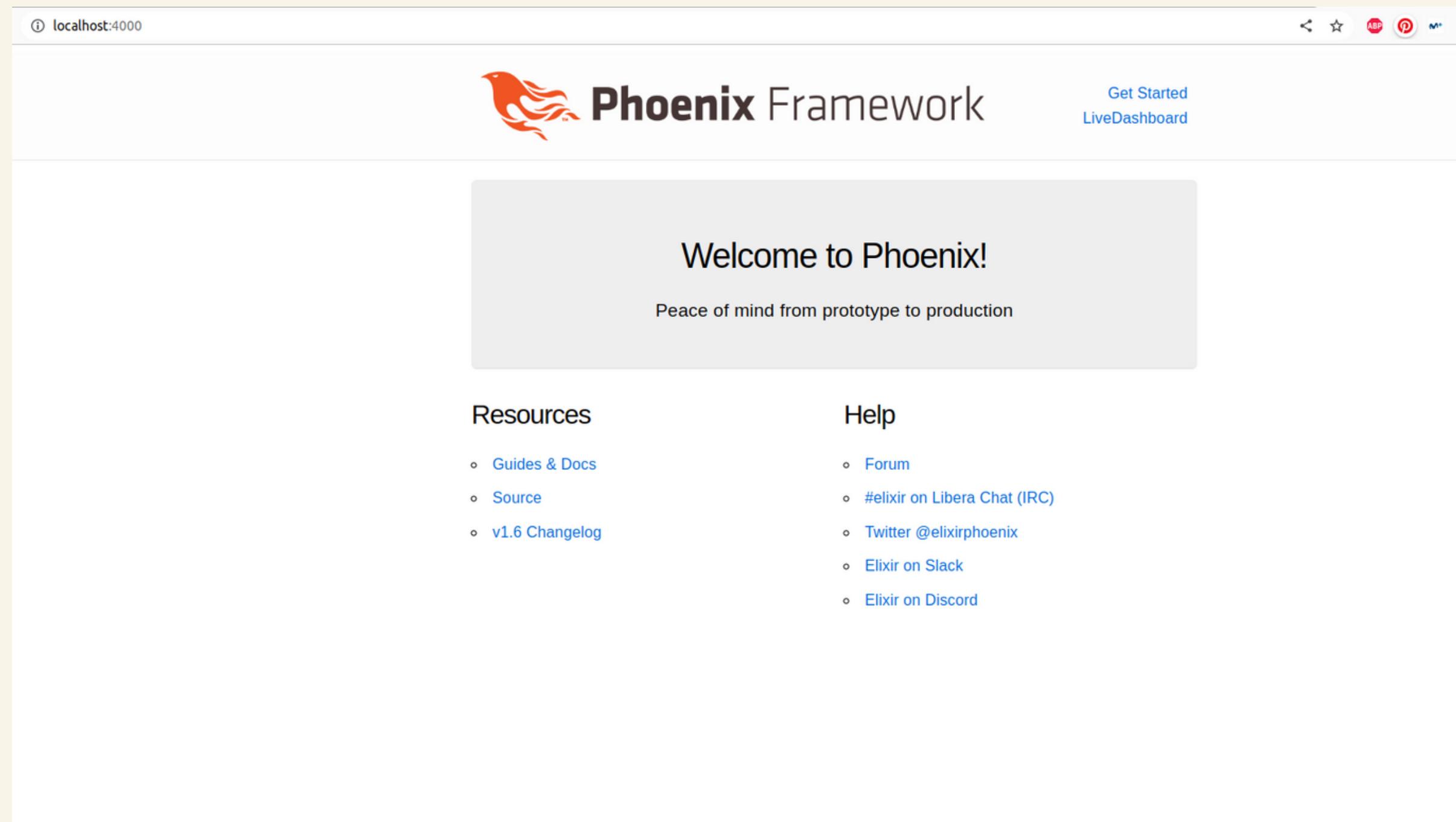


PRIMER PASO: INICIALIZAR EL PROYECTO

- Con unos comandos cómodos Phoenix nos permite inicializar una aplicación con nuestra configuración deseada.
- Opciones posibles:
 - --database: especificar el adaptador de ecto.
 - --no-html: no generar las vistas html.
 - --binary-id: usar binary_id como clave primaria.
- Todo adaptable usando `mix phx.gen.<keyword>`
- Usamos el comando para nuestro caso:

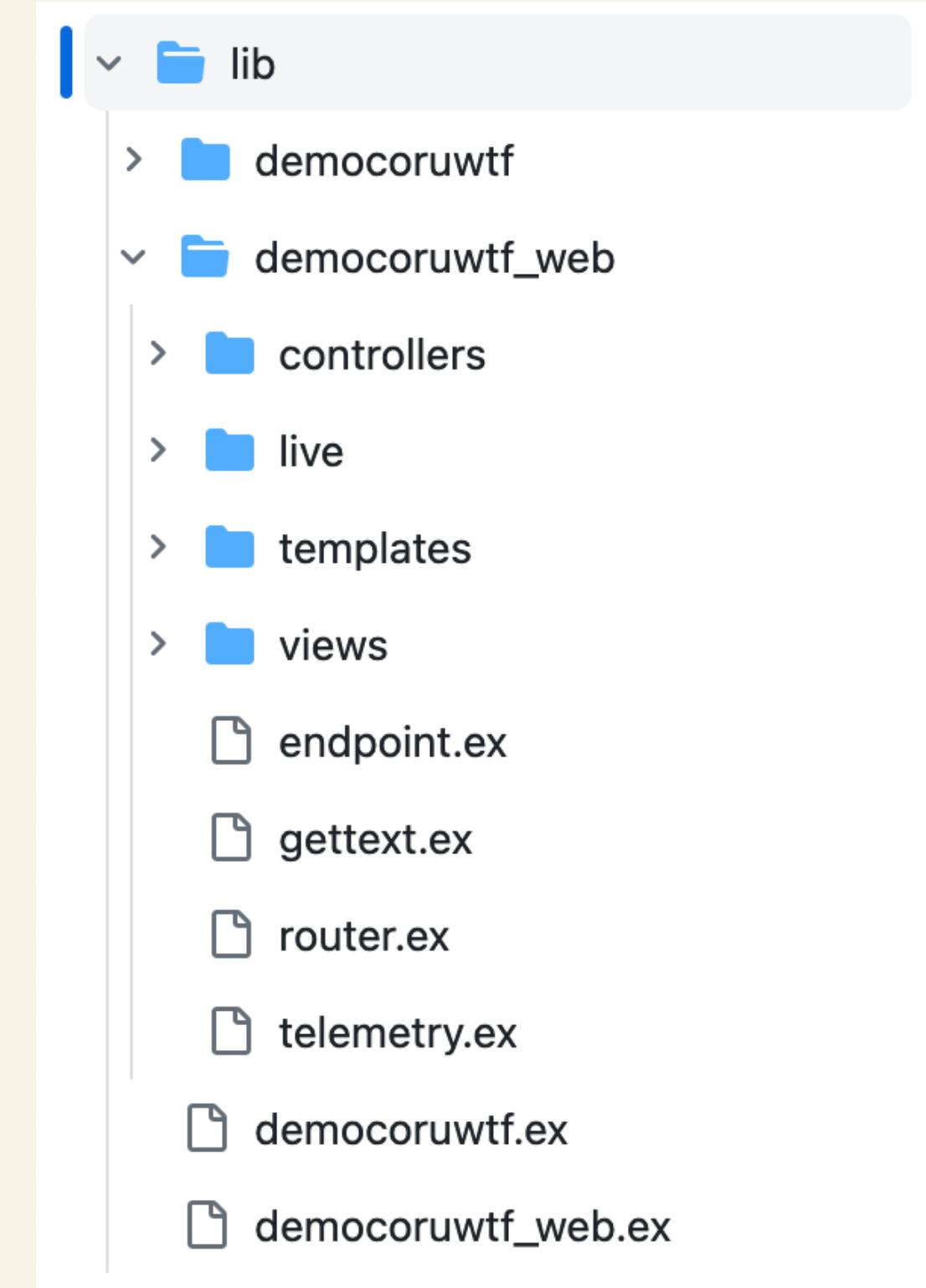
`mix phx.new democoruwtf`

RESULTADO:



ESTRUCTURA DEL PROYECTO

- El código está contenido dentro de lib.
- Dependencias en mix.exs (como un package.json)
- En la carpeta democoruwtf se encuentra todo el código correspondiente al modelo.
- En la carpeta _web está todo lo de vista y controlador.
- En router.ex está el mapeo de rutas y pies custom.
- En versiones más actualizadas del framework ya viene tailwind como dependencia.
- mix phx.routes -> ver todas tus rutas rápido



SEGUNDO PASO: CREAR NUESTRAS ENTIDADES

- Seguimos haciendo uso de los comandos de Phoenix.
- Generan todos los archivos en Modelo y vista necesarios.
- Añaden operaciones CRUD básicas funcionales.
- Las rutas del controlador deben ser añadidas en caso de querer cambiar el enrutamiento.
- Solo es preciso hacer cambios en restricciones, lógica de negocio o detalles visuales.

mix phx.gen.live Cards Card card content:string state:string

Context	Table name	Schema	Atributos de la entidad
mix phx.gen.live Cards Card	card	content:string state:string	

Qué queremos generar

BREAKING DOWN NUESTRA CARD

- phx.gen.live -> Indica a Phoenix que queremos generar LiveViews.
- Context -> API con las operaciones CRUD.
- Schema -> Definición del tipo y restricciones.

```
* creating lib/democoruwtf/cards/card.ex
* creating priv/repo/migrations/20230910224942_create_card.exs
* creating lib/democoruwtf/cards.ex
* injecting lib/democoruwtf/cards.ex
* creating test/democoruwtf/cards_test.exs
* injecting test/democoruwtf/cards_test.exs
* creating test/support/fixtures/cards_fixtures.ex
* injecting test/support/fixtures/cards_fixtures.ex
* injecting lib/democoruwtf_web.ex

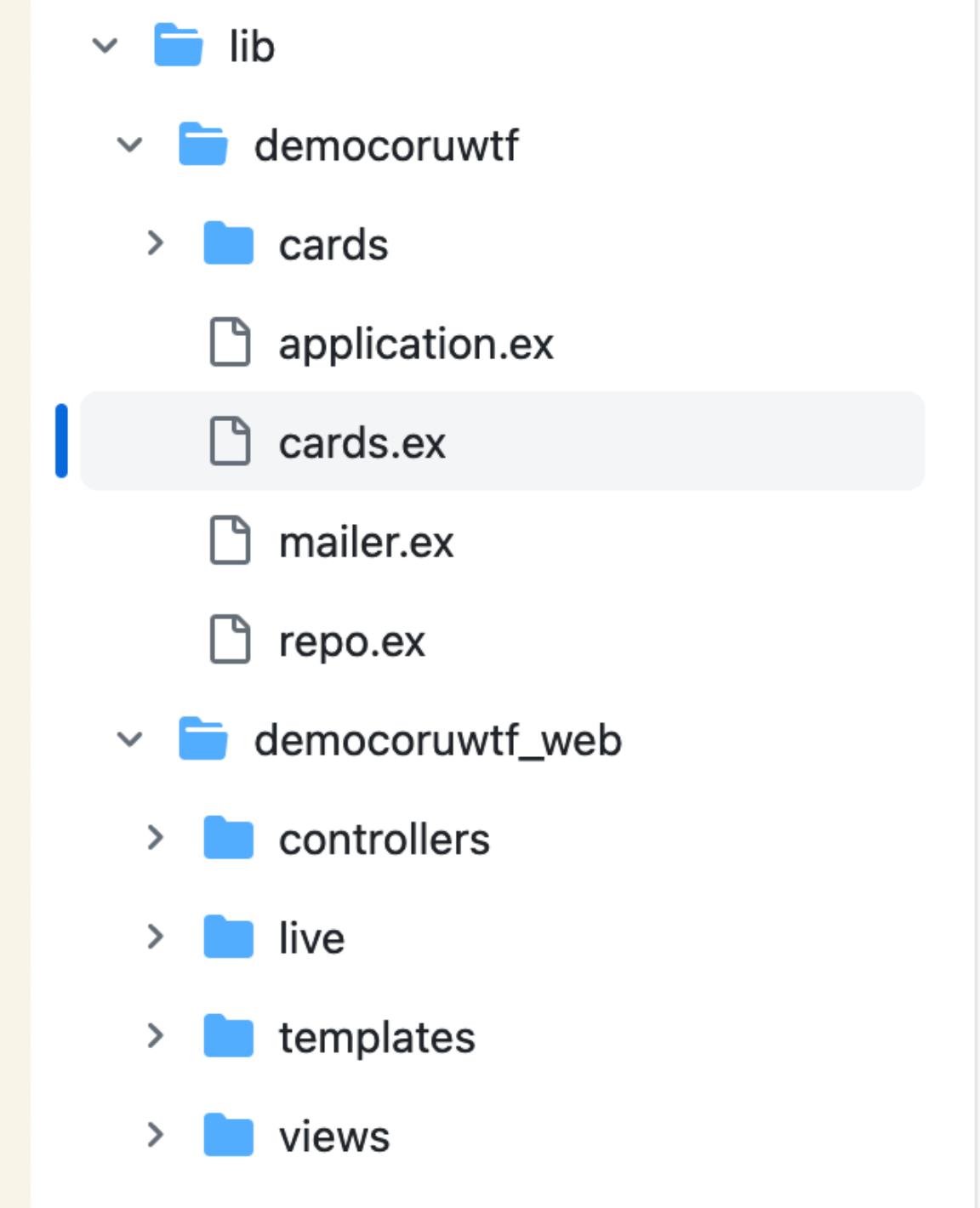
Add the live routes to your browser scope in lib/democoruwtf_web/router.ex:

  live "/card", CardLive.Index, :index
  live "/card/new", CardLive.Index, :new
  live "/card/:id/edit", CardLive.Index, :edit

  live "/card/:id", CardLive.Show, :show
  live "/card/:id/show/edit", CardLive.Show, :edit

Remember to update your repository by running migrations:

$ mix ecto.migrate
```



BREAKING DOWN NUESTRA CARD

- Un schema define el tipo de dato con el que se trabajará.
- Define restricciones a los campos.
- La función changeset se encarga de comprobar que sea un struct válido.

```
defmodule Democoruwtf.Cards.Card do
  use Ecto.Schema
  import Ecto.Changeset

  schema "card" do
    field :content, :string
    field :state, :string

    timestamps()
  end

  @doc false
  def changeset(card, attrs) do
    card
    |> cast(attrs, [:content, :state])
    |> validate_required([:content, :state], message: "Todos los campos son obligatorios")
    |> validate_inclusion(:state, ["todo", "in progress", "done"])
  end
end
```

BREAKING DOWN NUESTRA CARD

- Ejemplo de la API que se nos genera con estos comandos, que sirve como contexto.
- Repo es un módulo que nos permite trabajar con la base de datos abstrayéndonos de detalles.

```
@doc """
Creates a card.

## Examples

iex> create_card(%{field: value})
{:ok, %Card{}}

iex> create_card(%{field: bad_value})
{:error, %Ecto.Changeset{}}

"""

def create_card(attrs \\ %{}) do
  %Card{}
  |> Card.changeset(attrs)
  |> Repo.insert()
end
```

BREAKING DOWN NUESTRA CARD

- Phoenix nos permite abstraernos de la base de datos.
- Para ello, no trabajamos con scripts, SQL o noSQL.
- Para crear la tablas, alterarlas o borrarlas se usan migraciones.
- Localizadas en priv/repo/migrations

```
defmodule Democoruwtf.Repo.Migrations.CreateCard do
  use Ecto.Migration

  def change do
    create table(:card) do
      add :content, :string
      add :state, :string
      timestamps()
    end
  end
end
```

TERCER PASO: AÑADIR RESTRICCIÓN A LA CARD

- Un detalle que tenía la card era que solo aceptaría en el estado tres opciones: Todo, In progress, Done.
- Es algo que afecta a todas las operaciones, el comprobar ese valor.
- Pertenece a la definición del esquema.
- Las comprobaciones y errores al respecto las gestiona changeset/2

```
@doc false
def changeset(card, attrs) do
  card
    |> cast(attrs, [:content, :state])
    |> validate_required([:content, :state], message: "campos obligatorios")
    |> validate_inclusion(:state, ["todo", "in progress", "done"], message: "estado inválido")
end
```

CUARTO PASO: ARRANCAMOS LA APP

Phoenix.Ecto.PendingMigrationError at GET /

there are pending migrations for repo: Democoruwtf.Repo. Try running `mix ecto.migrate` in the command line to migrate it



Run migrations for repo

lib/phoenix_ecto/check_repo_status.ex

```
62      {:ok, migration_directories, migrations} ->
63          has_pending =
64              Enum.any?(migrations, fn {status, _version, _migration} -> status == :down end)
65
66      if has_pending do
67          raise Phoenix.Ecto.PendingMigrationError, repo: repo, directories: migration_directories
68      else
69          false
70      end
71
72  :error ->
```

Phoenix.Ecto.CheckRepoStatus.check_pending_migrations!/2

phoenix_ecto

Show only app frames

- phoenix_ecto - lib/phoenix_ecto/check_repo_status.ex:67
- phoenix_ecto - lib/phoenix_ecto/check_repo_status.ex:31
- elixir - lib/enum.ex:2396
- phoenix_ecto - lib/phoenix_ecto/check_repo_status.ex:30
- lib/democoruwtf_web/endpoint.ex:1
- lib/plug/debugger.ex:136
- lib/democoruwtf_web/endpoint.ex:1
- phoenix - lib/phoenix/endpoint/cowboy2_handler.ex:54
- cowboy - /home/carla/charlas/demo-coruwtf/deps/cowboy/src/cowboy_handler.erl:37
- cowboy - /home/carla/charlas/demo-coruwtf/deps/cowboy/src/cowboy_stream_h.erl:306
- cowboy - /home/carla/charlas/demo-coruwtf/deps/cowboy/src/cowboy_stream_h.erl:295
- stdlib - proc_lib.erl:226

Copy markdown

Phoenix.Ecto.CheckRepoStatus.check_pending_migrations!/2
anonymous fn/3 in Phoenix.Ecto.CheckRepoStatus.call/2
Enum.-"reduce/3-lists^fold/2-0-"/3
Phoenix.Ecto.CheckRepoStatus.call/2
DemocoruwtfWeb.Endpoint.plug_builder_call/2
DemocoruwtfWeb.Endpoint."call (overridable 3)"/2
DemocoruwtfWeb.Endpoint.call/2
Phoenix.Endpoint.Cowboy2Handler.init/4
:cowboy_handler.execute/2
:cowboy_stream_h.execute/3
:cowboy_stream_h.request_process/3
:proc_lib.init_p_do_apply/3

► Request info

► Headers

LA LIVEVIEW DE NUESTRA CARD



Phoenix Framework

[Listing Card](#)

Content	State	
hola numero 2	todo	Show Edit Delete

[New Card](#)

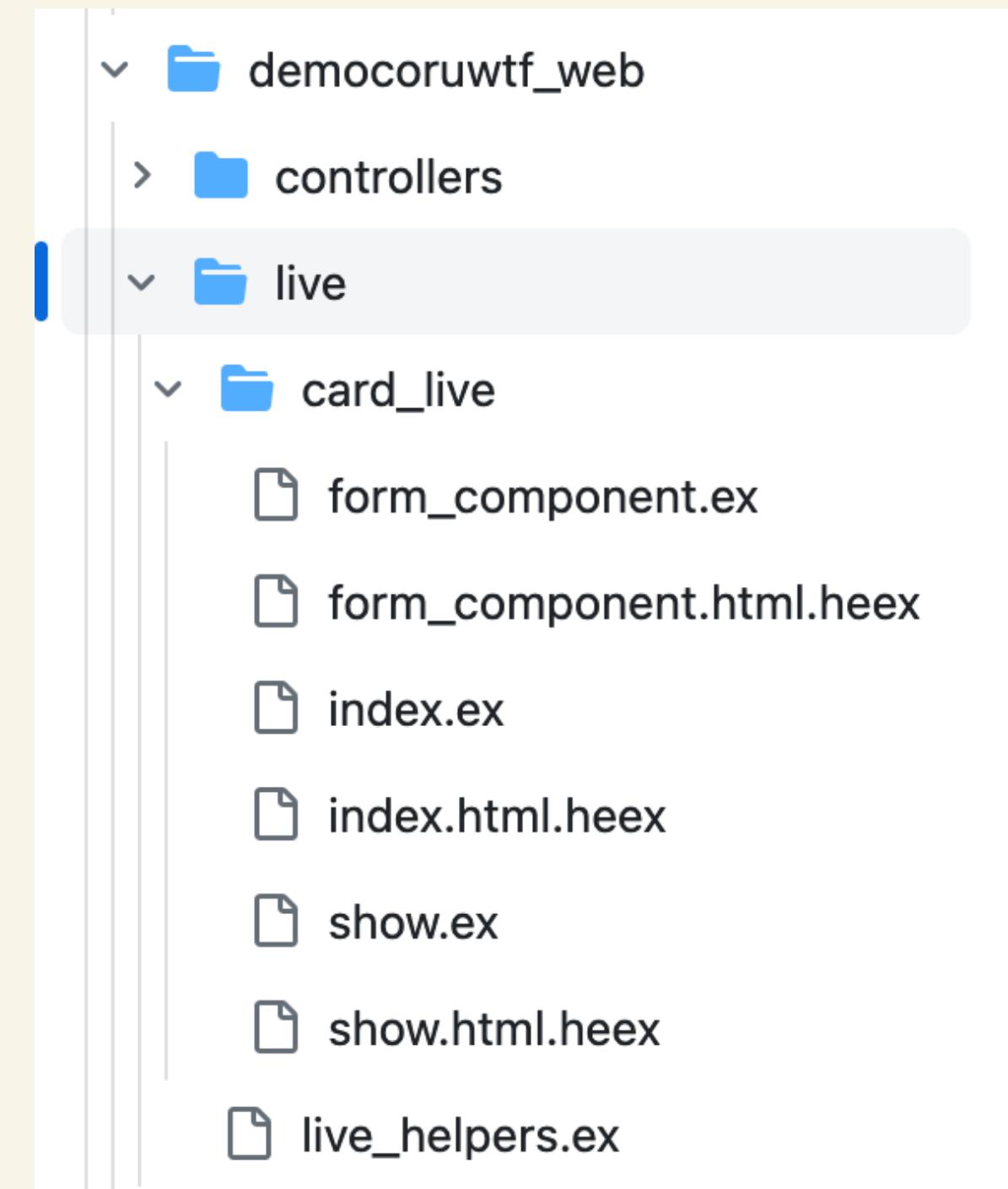
En localhost:4000/card

¿QUÉ ES LIVEVIEW?

- Librería de Elixir/Phoenix para implementar aplicaciones interactivas sin usar Javascript.
- Crea aplicaciones SPA, el servidor manda los cambios al navegador.
- Los eventos de LiveView son típicos mensajes de Elixir.
- Nos da herramientas para crear componentes con su lógica propia.
- Los componentes de LiveView tienen un ciclo de vida especial.
- La comunicación servidor navegador va a través de un websocket, donde se guarda y manipula el estado.

NUESTRO LIVEVIEW

- Lo que nos genera son todas las vistas en LiveView necesarias para trabajar con las operaciones CRUD.
- Los archivos .ex son lógica y .heex la plantilla html.
- En los archivos de lógica encontramos las operaciones CRUD correspondientes a ese componente y funciones handle_event.
- index.es -> LiveView
- form_component -> LiveComponent



PASO CINCO: APLICAMOS MAGIA DEL CSS



Phoenix Framework

Get Started
LiveDashboard

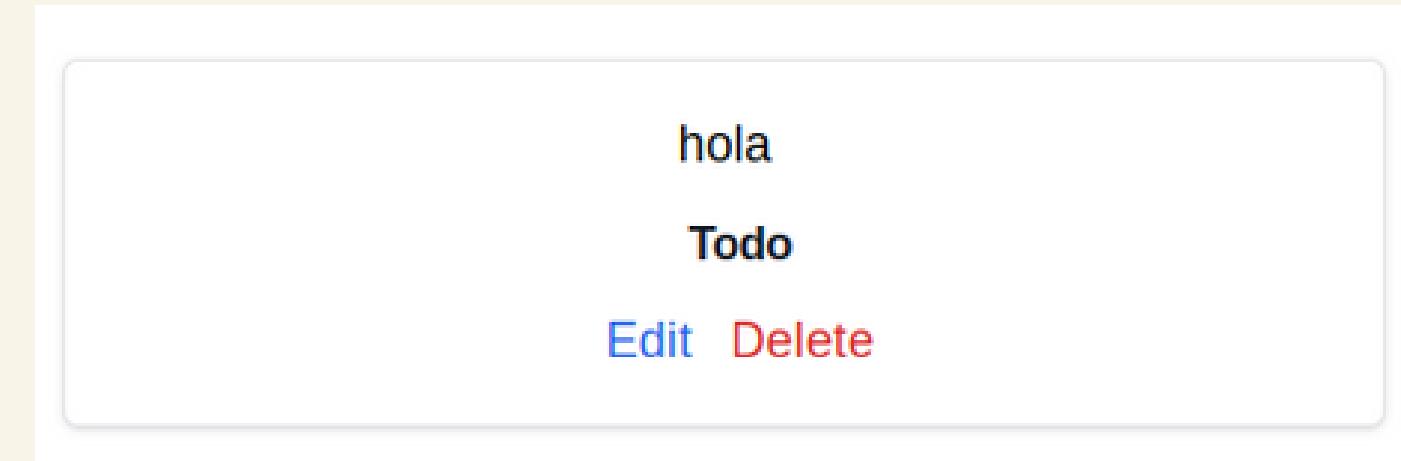
Our kanban

Content	State	
card con contenido	todo	Edit Delete
hola	todo	Edit Delete
hola 4	todo	Edit Delete

New Card

PASO SEIS: CONVERTIR NUESTRA LISTA EN TARJETAS DE KANBAN

- Descomponer el tablero en tarjetas.
- Creamos un LiveComponent aunque no tenga comportamiento real time.
- Encapsula toda la lógica, estilos y código html.
- Recibe la card y esta se guardará en los assigns del componente.



```
<tr class='hover:bg-gray-100' id={"card-#{card.id}">
  <.live_component module={DemocoruwtfWeb.CardLive.Card} id={"card-#{card.id}">} card={card} rounded>

</tr>
```

COMO SE VERÍA NUESTRO MÓDULO CARD (CON TAILWIND)

```
defmodule DemocoruwtfWeb.CardLive.Card do
  @moduledoc """
  A sample component generated by `mix surface.init`.
  """

  use Phoenix.LiveComponent
  use Phoenix.HTML

  alias Democoruwtf.Cards
  alias Democoruwtf.Cards.Card

  @impl true
  def render(assigns) do
    ~H"""
    <div class={"block mx-12 p-6 bg-white border border-gray-200 rounded-lg shadow hover:bg-gray-100 space-x-4 space-y-4 my-8"}>
      <div class="text-center mx-8 "><%= @card.content %></div>
      <h5 class="mb-2 text-2xl font-bold tracking-tight text-gray-900 text-center capitalize" ><%= @card.state %></h5>

      <div class="flex justify-center items-center flex-row space-x-5">
        <span class="inline-flex items-center font-medium text-blue-600 dark:text-blue-500 hover:underline"><%= live_patch "Edit", to: DemocoruwtfWeb.Router.Helpers.card_index_path(@socket, :edit, @card) %></span>
        <span class="inline-flex items-center font-medium text-red-600 dark:text-red-600 hover:underline"><%= link "Delete", to: "#", phx_click: "delete", phx_value_id: @card.id, data: [confirm: "Are you sure?"] %></span>
      </div>
    </div>
  """
  end
end
```



Reacción normal

COMO SE VERÍA NUESTRO MÓDULO CARD (SIN TAILWIND)

```
defmodule DemocoruwtfWeb.CardLive.Card do
  @moduledoc """
  A sample component generated by `mix surface.init`.
  """

  use Phoenix.LiveComponent
  use Phoenix.HTML

  alias Democoruwtf.Cards
  alias Democoruwtf.Cards.Card

  @impl true
  def render(assigns) do
    ~H"""
    <div>
      <div><%= @card.content %></div>
      <h5><%= @card.state %></h5>

      <div>
        <span><%= live_patch "Edit", to: DemocoruwtfWeb.Router.Helpers.card_index_path(@socket, :edit, @card) %></span>
        <span><%= link "Delete", to: "#", phx_click: "delete", phx_value_id: @card.id, data: [confirm: "Are you sure?"] %></span>
      </div>
    </div>
    """
  end
end
```



Reacción normal
backender

AGRUPAMOS POR COLUMNAS

```
<div class="grid grid-cols-3 gap-4 my-24">
<div>
  <h2 class="text-center text-4xl font-bold dark:text-white">Todo</h2>
  <%= for card <- @card_collection do %>
    <%= if card.state == "todo" do %>
      <tr class='hover:bg-gray-100' id={"card-#{card.id}">
        <.live_component module={DemocoruwtfWeb.CardLive.Card} id={"card-#{card.id}"> card={card} rounded/>
      </tr>
    <% end %>
  <% end %>
</div>
```

- Por simplificar, hacemos 3 bucles para cada uno de los estados de la tarjeta.
- La estructura se realiza con las clases de tailwind.
- Posibles mejoras añadiendo funciones para recolectar las cards según el estado.

AGRUPAMOS POR COLUMNAS



Phoenix Framework

[Get Started](#)
[LiveDashboard](#)

Card created successfully

Our kanban

Todo

hola

Todo

[Edit](#) [Delete](#)

hola 4

Todo

[Edit](#) [Delete](#)

In Progress

vambiar tipo de campo dentro del state

In Progress

[Edit](#) [Delete](#)

Done

card con contenido pero bastante largo como
para asustar a la gente

Done

[Edit](#) [Delete](#)

New Card

EJEMPLO



Phoenix Framework

[Get Started](#)
[LiveDashboard](#)

Card created successfully

Our kanban

Todo

card 1

Todo

[Edit](#) [Delete](#)

card 2

Todo

[Edit](#) [Delete](#)

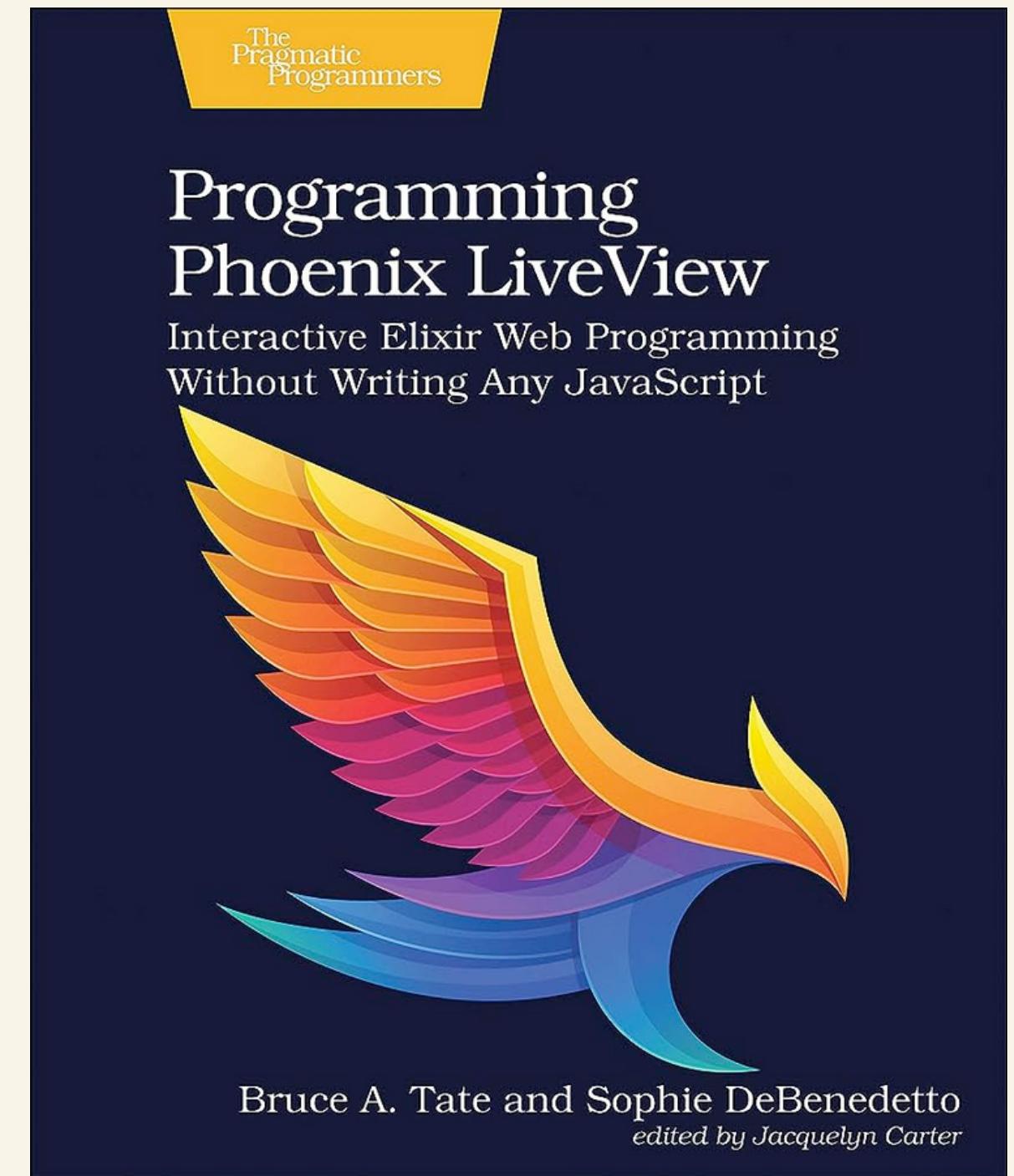
In Progress

Done

[New Card](#)

PASO SIETE: AÑADIR COMPORTAMIENTO REAL-TIME

- Además de poder crear componentes, páginas y controlar eventos de los usuarios, podemos controlar eventos en tiempo real.
- Elixir nos proporciona funciones como `handle_info/2`.
- `handle_info/2` espera a mensajes a través del socket.
- Desde el modelo podemos enviar mensajes y emplear métodos PubSub.



CAMBIOS EN EL MODELO

- En el modelo se suscriben los clientes.
- Se envían desde aquí los mensajes cada vez que cambie algo en el modelo.
- El módulo PubSub de Phoenix implementa este comportamiento.

```
def subscribe do
  Phoenix.PubSub.subscribe(Democoruwtf.PubSub, "cards")
end

defp broadcast({:error, _reason} = error, _event), do: error

defp broadcast({:ok, post}, event) do
  Phoenix.PubSub.broadcast(Democoruwtf.PubSub, "cards", {event, post})
  {:ok, post}
end
```

CAMBIOS EN EL MODELO

- Para cada función que suponga un cambio importante hace un broadcast.
- Un broadcast es un mensaje a todos los subscriptores.
- |> es el pipe operator de Elixir.

```
@doc """
Creates a card.

## Examples

    iex> create_card(%{field: value})
    {:ok, %Card{}}

    iex> create_card(%{field: bad_value})
    {:error, %Ecto.Changeset{}}

    ...
    def create_card(attrs \\ %{}) do
      %Card{}
      |> Card.changeset(attrs)
      |> Repo.insert()
      |> broadcast(:card_created)
    end
```

CAMBIOS EN LA VISTA

E

- La función `handle_info/2` reciben el mensaje del modelo.
- En el mensaje se recibe el tipo de evento y la card.
- Desde esa función se puede acceder al socket y modificar el estado.
- Con la función `update` se cambia el estado en el socket y se renderiza de nuevo la vista.

```
@impl true

def handle_info({:card_updated, card}, socket) do
  list = Enum.filter(socket.assigns.card_collection, fn el -> el.id != card.id end)

  {:noreply,
   update(socket, :card_collection, fn card ->
     [card | list]
   end)}
end
```

EJEMPLO

The screenshot shows a Phoenix Framework application interface. At the top left is a red bird logo and the text "Phoenix Framework". To the right are links for "Get Started" and "LiveDashboard". Below this is a section titled "Our kanban" with three columns: "Todo", "In Progress", and "Done".

- Todo:** Contains two cards:
 - card 1: Todo status, Edit, Delete buttons.
 - card 4: Todo status, Edit, Delete buttons.
- In Progress:** Contains one card:
 - Card 3: In Progress status, Edit, Delete buttons.
- Done:** Contains no visible cards.

A green "New Card" button is located at the bottom left of the kanban area.

The screenshot shows the same Phoenix Framework application interface after an action. A green banner at the top says "Card created successfully". The "Our kanban" section now has four columns: "Todo", "In Progress", "Done", and a new "New" column.

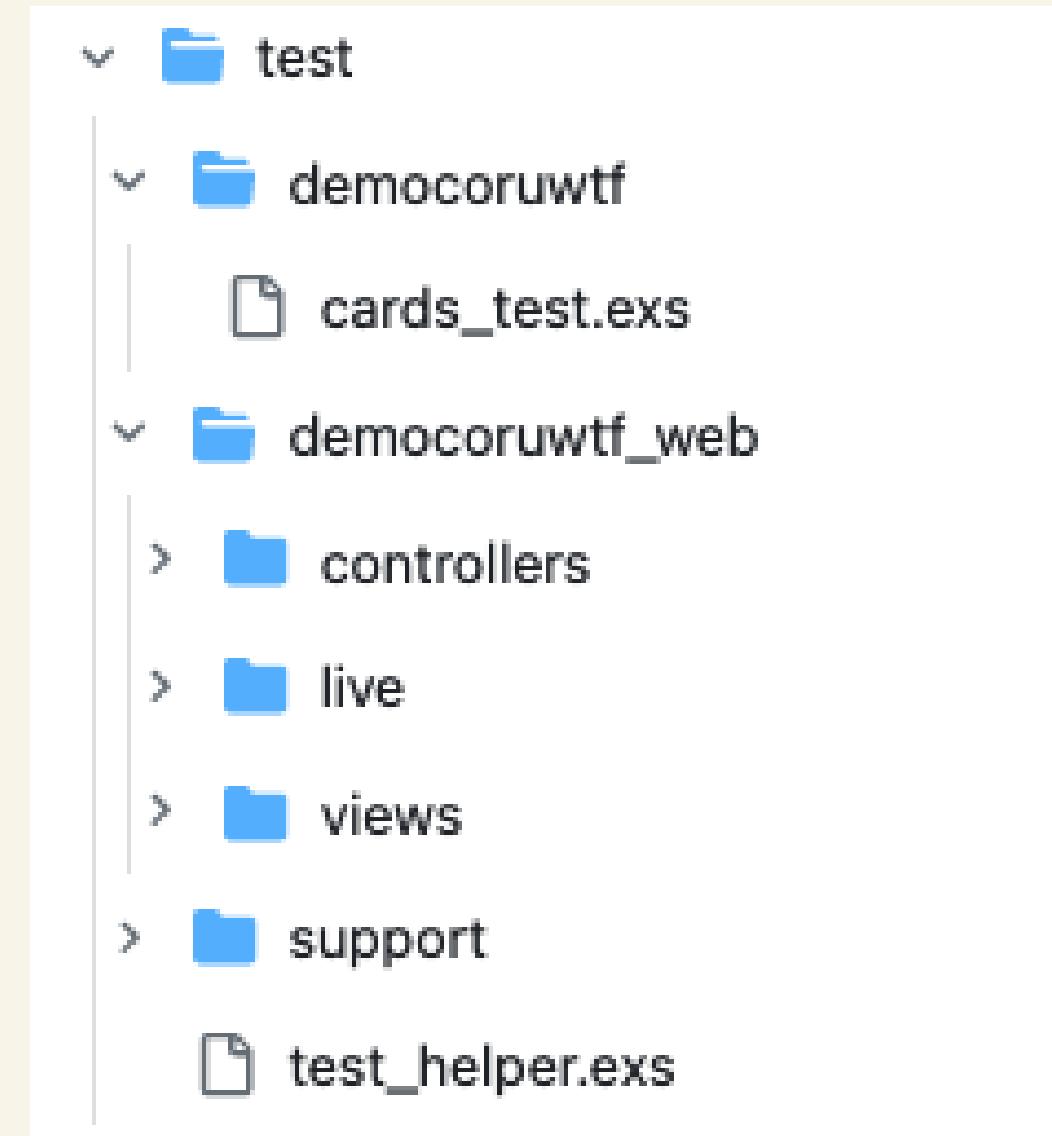
- Todo:** Contains two cards:
 - card 1: Todo status, Edit, Delete buttons.
 - card 4: Todo status, Edit, Delete buttons.
- In Progress:** Contains one card:
 - Card 3: In Progress status, Edit, Delete buttons.
- Done:** Contains no visible cards.
- New:** Contains one card:
 - card 5: New status, Edit, Delete buttons.

A green "New Card" button is located at the bottom right of the kanban area.

**¿QUÉ FALTA EN TODO ESTE
PROCESO?**

TESTS

- No nos tenemos que preocupar.
- Phoenix nos los crea, pone una estructura y da unos ejemplos base reutilizables.
- Existen tests para modelo y todo tipo de vista que queramos implementar.
- Con mix test --cover obtenemos un análisis de los tests.
- En esta charla no nos tenemos que preocupar :)



MISCELLANIOUS

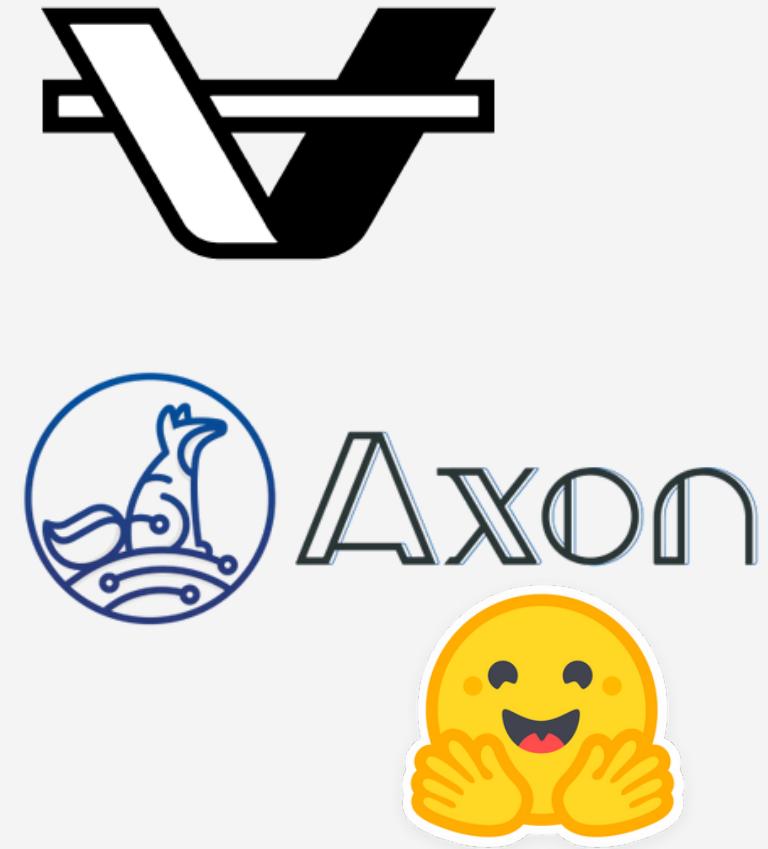
- Para desplegar una aplicación en Phoenix -> Gigalixir
- Github actions permite automatizar procesos como pasar tests, tener un coverage mínimo, desplegar...
- Herramientas de control de la calidad como dialyxir.
- Facilidad para implementar funcionalidades como autenticación, autorización o envío de emails.
- Gran variedad de librerías gracias a la comunidad, por ejemplo, paginación -> scrivener_ecto.

Elixir para todos



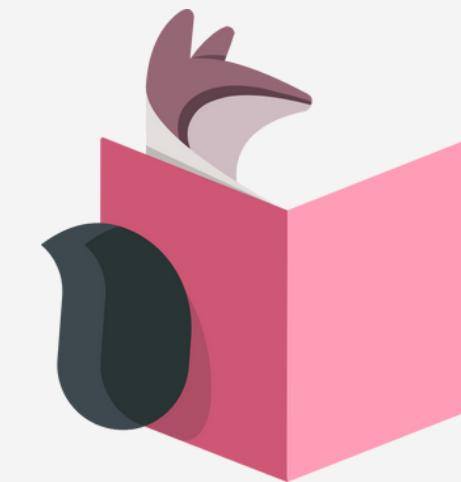
A PARTE DE PHOENIX

- Además de crear aplicaciones web con Phoenix existe Ash Framework.
- También se está trabajando para desarrollo de aplicaciones nativas de móvil -> LiveView native.
- Trabajar con redes neuronales en Elixir -> Axon (<https://dockyard.com/blog/2022/01/11/getting-started-with-axon>)
- Trabajar con arquitecturas -> Procesos y Genservers.



A PARTE DE PHOENIX

- Trabajar solo con APIs -> Se pueden generar aplicaciones sin liveview, generando vistas JSON.
- Bases de datos no relacionales -> Existen drivers para manejarse.
- LiveBook -> Un proyecto muy nuevo que sirve para documentar, desplegar aplicaciones, visualizar datos, trabajar con modelos de machine learning.
- Gráficos y data visualization aprovechando liveview-> Paquete context (<https://github.com/mindok/context>)
- Phoenix + React -> <https://blog.logrocket.com/to-do-list-phoenix-react-typescript/>



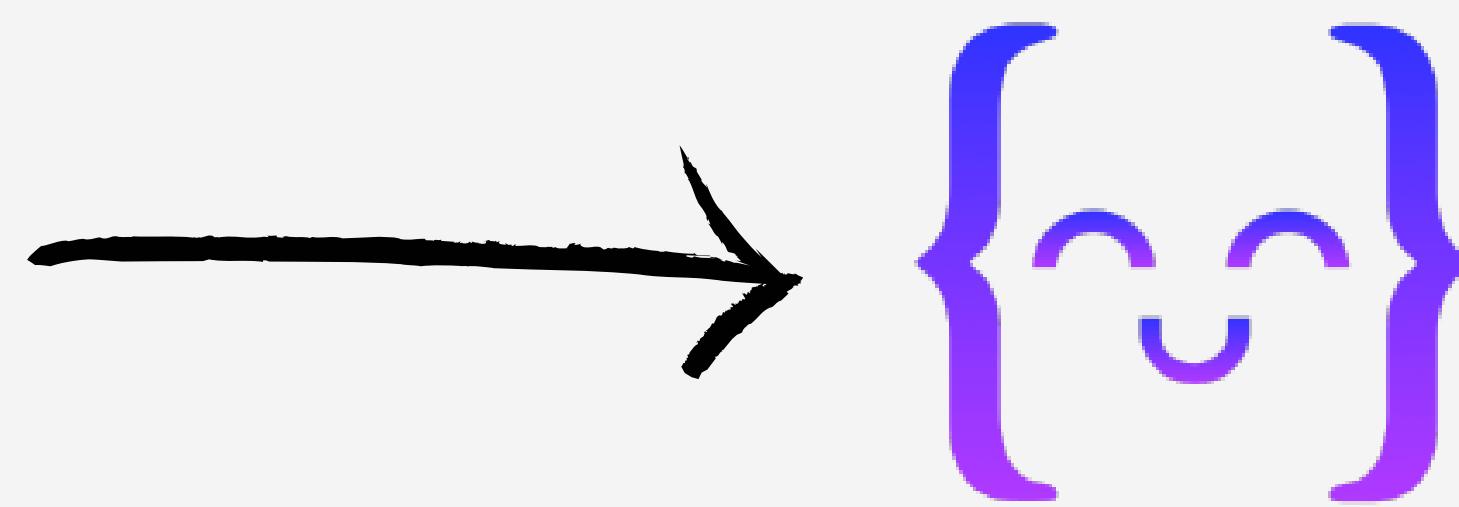
TUTORIAL RECONOCER DÍGITOS



IMAGE CLASSIFICATION



PARA EVITAR ESTA REACCIÓN



Linkedin



Gracias

Repositorio

