SNGULAR

Phoenix, una alternativa para el desarrollo web con Elixir

Carla Rodríguez Estévez

Conociendo el lenguaje

- Lenguaje de programación funcional y dinámico.
- Compatibilidad con Erlang y la Beam.
- Escalable y tolerante a fallos.
- Creado para manejar concurrencia y paralelismo cómodamente.
- Inmutabilidad: crea nuevos datos basados en los datos originales.
- Diseñado por José Valim

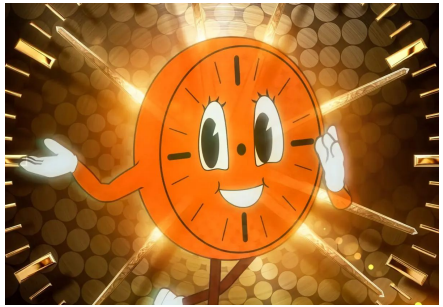
¿Dónde lo podemos ver?

- Discord: En su backend y sistemas de mensajería en tiempo real.
- Spotify: En desarrollo backend.
- Cabify: Desarrollo backend principalmente.
- PepsiCo: En los equipos de Search Marketing y Sales Intelligence Platform para herramientas internas.
- Slack: En su servidor de media para p2p y llamadas grupales.
- ....

Unas pequeñas pinceladas iniciales

```
iex> 255 #Enteros
iex> 3.14 #Coma flotante
iex> true || false #Booleanos
iex> "Hello World" #Cadenas
iex> :foo == :bar #Átomos
```

Tipos básicos



Lo fundamental en el tiempo que hay

Operaciones básicas

```
iex> 2 + 2 #Aritméticas
iex> #Booleanas
iex> -20 || true
-20
iex> false || -42
42
iex> 42 && true
true
iex> 42 && nil
nil
iex> !42
false
iex> 42 and true   #Primer valor debe ser booleano
** (ArgumentError) argument error: 42
iex> not 42
** (ArgumentError) argument error
iex> #Comparaciones
iex> 2 == 2.0
true
iex> 2 === 2.0
false
```

Colecciones

Listas

```
iex> [3.14, :pie, "Apple"]

        Agregar elemento
iex> list = [3.14, :pie, "Apple"]
[3.14, :pie, "Apple"]
iex> ["π" | list]
["π", 3.14, :pie, "Apple"]

        Concatenación
iex> [1,2] ++ [3,4,1]
[1,2,3,4,1]

        Cabeza/cola
iex> hd [3.14, :pie, "Apple"]
3.14
iex> tl [3.14, :pie, "Apple"]
[:pie, "Apple"]
```

Listas de palabras clave

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

Lista de tuplas, donde el primer elemento es un átomo.
Las claves están ordenadas y pueden no ser únicas.

Mapas

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

iex> %{map | foo: "baz"}    #Actualizar clave
%{foo: "baz", hello: "world"}
```

Pattern matching

Operador de coincidencia

```
1
2  # Listas
3
4  [1 | tail] = list
5  [1, 2, 3]
6  tail
7  [2, 3]
8  [2|_] = list
9  ** (MatchError) no match of right hand side value: [1, 2, 3]
10
11 # Tuplas
12 iex> {:ok, value} = {:ok, "Successful!"}
13 {:ok, "Successful!"}
14 iex> value
15 "Successful!"
16 iex> {:ok, value} = {:error}
17 ** (MatchError) no match of right hand side value: {:error}
18
19
20
```

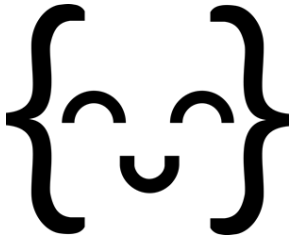
Funciones y pattern matching

```
1  # Pattern matching
2
3  iex>   def b_not(true), do: false
4  iex>   def b_not(false), do: true
5
6  iex>   def b_and(true,true),do: true
7  iex>   def b_and(_,_), do: false
8
9  iex>   def b_or(false,false),do: false
10 iex>   def b_or(_,_),do: true
11
12 #Pipe operator
13 iex> pan |> tostar(minutos: 2) |> poner_mermelada |> comer
14
```

```
14
15 #Módulos y funciones
16
17 defmodule Greeter do
18   def hello(names) when is_list(names) do.  #Guardia
19     names = Enum.join(names, ", ")
20
21     hello(names)
22   end
23
24   def hello(name) when is_binary(name) do
25     phrase() <> name
26   end
27
28   defp phrase, do: "Hello, "       #Función privada
29 end
30
```

Más allá de lo básico

- Herramienta Mix.
- Especificación de tipos y herramientas de QA.
- Behaviours.
- Procesos y concurrencia en elixir.
- Supervisores, link de procesos, GenServers...
- Distribución y nodos.




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

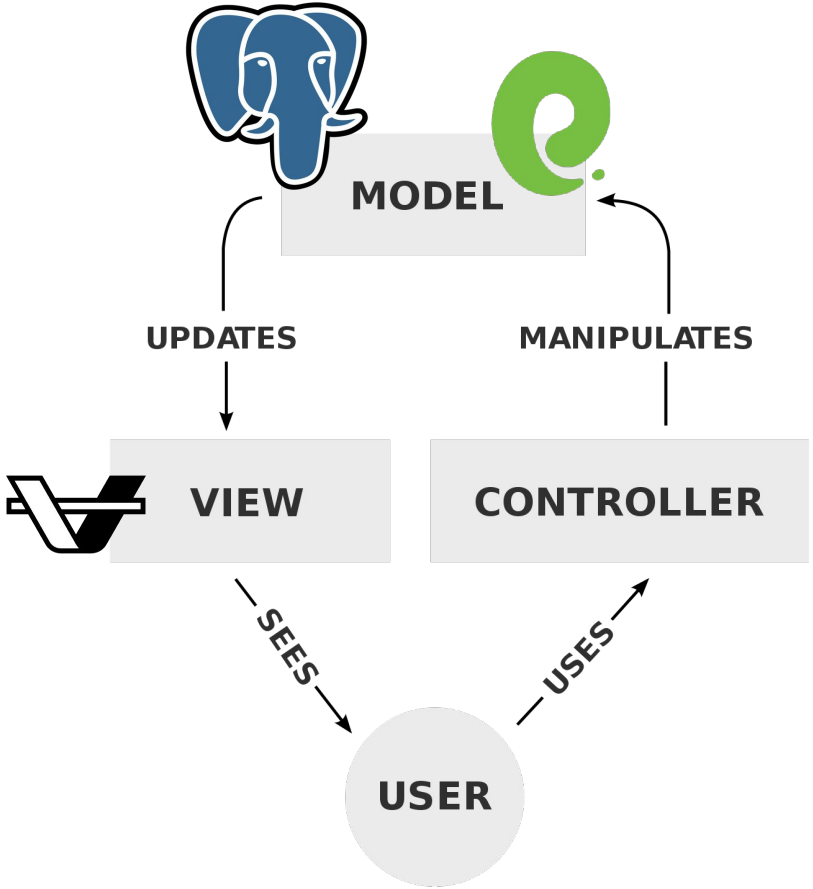
spawn -> crea un proceso y devuelve su PID.
send -> envía un mensaje al PID indicado.
receive -> bloquea el proceso hasta recibir un mensaje.

¿Qué es Phoenix?

- Framework de desarrollo web.
- Arquitectura Model-View-Controller
- Abstrae al programador de detalles técnicos para centrarse en lógica de negocio.
- Aprovecha las ventajas de elixir en concurrencia y paralelismo.
- Liveview: Permite crear aplicaciones en tiempo real.
- Soporte para websockets.
- Creado por Chris McCord.

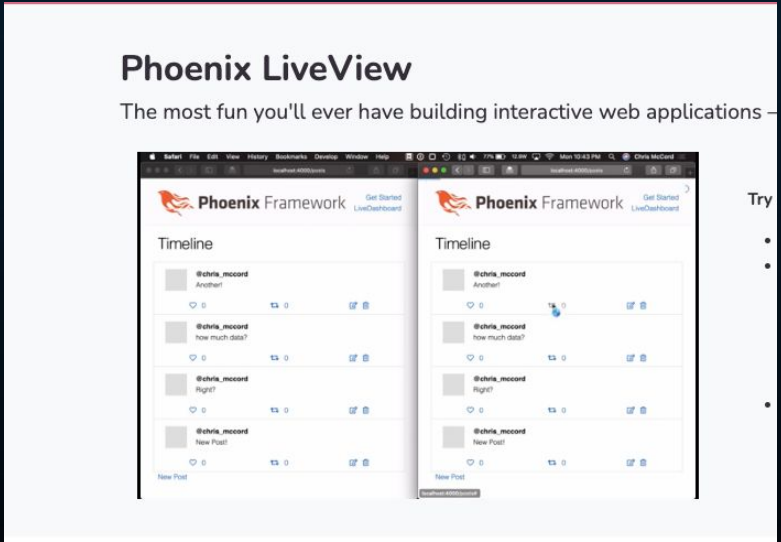
Estructura de una aplicación

- Arquitectura MVC, aunque no típica.
- **Modelo**: Se encarga de la persistencia y mantener la lógica de negocio. Compatible con varias BDs.
- **Controlador**: Mapea rutas y peticiones con los módulos adecuados. Gestiona pipes.
- **Vista**: Soporta eventos en tiempo real, creación de SPAs y varios tipas de vista a la vez.



Demo

Chat en tiempo real - Twitter clone

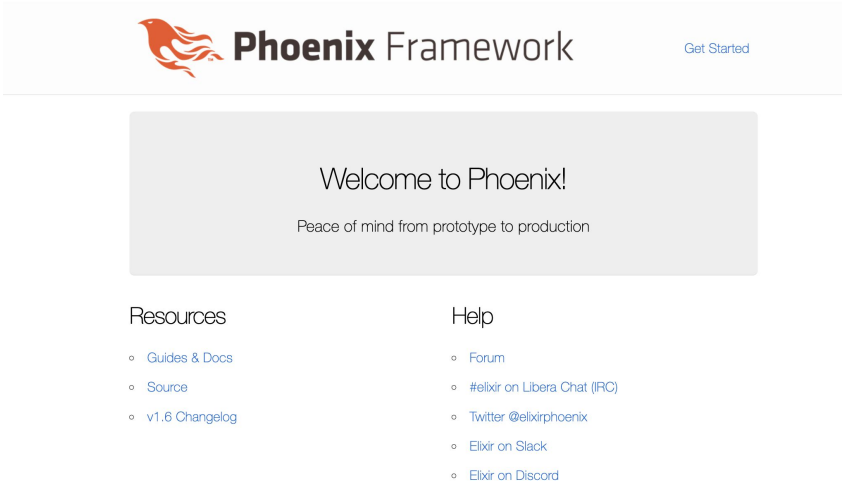


START-UP MINDSET

We never lose our restless curiosity and energy.

1º Paso - Inicializar proyecto

mix phx.new demoatlant



- Phoenix nos proporciona unos comandos de generación de código muy cómodos.
- Ejemplos: --database, --no-html.....
- Todo adaptable con *mix phx.gen.<keyword>*.

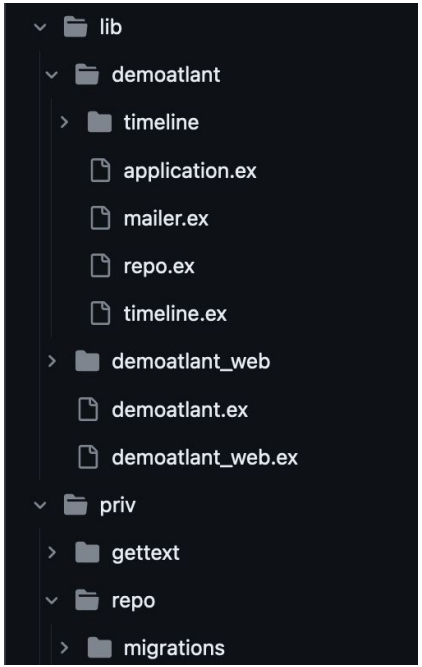
2º Paso - Crear nuestras entidades

```
mix phx.gen.live Timeline Post posts username:string content:string likes_count:integer
repost_count:integer
```

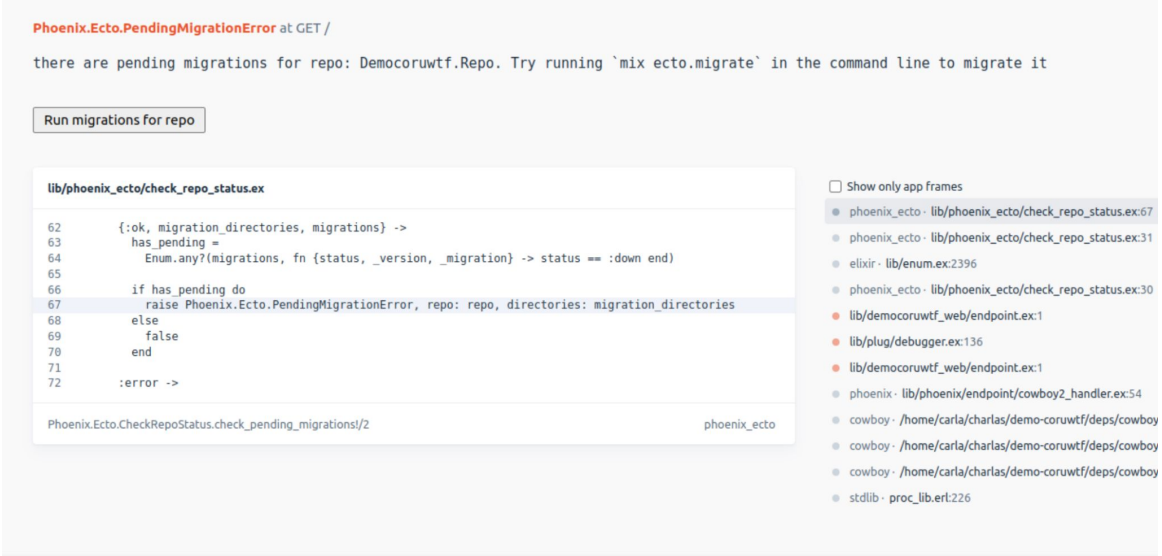
- **phx.gen.live** -> Indica a Phoenix que queremos generar LiveViews.
- **Timeline** -> Context, API con las operaciones CRUD básicas.
- **Post** -> Schema, definición del tipo y restricciones.
- Username, content… -> Atributos de la entidad.
- Las rutas en el controlador deben añadirse como indica la consola.
- Solo es preciso hacer cambios en la lógica de negocio o ampliar restricciones.

2º Paso - Crear nuestras entidades

Nuestro proyecto inicial



Consejo: siempre hacer caso a nuestra terminal



3º Paso - Añadir restricciones

- Los schemas definen el tipo de dato con el que se trabajará.
- Define las restricciones de las entidades.
- *changeset/2* comprueba la validez del struct según la entidad definida.

- Los tipos que se añadan o modifiquen solo pedirán los campos username y body obligatoriamente.
- *validate_length/3* se asegura de no pasarnos con los caracteres.

```elixir
defmodule Demoatlant.Timeline.Post do
  use Ecto.Schema
  import Ecto.Changeset

  schema "posts" do
    field :body, :string
    field :likes_count, :integer, default: 0
    field :repost_count, :integer, default: 0
    field :username, :string, default: "Carla"

    timestamps()
  end

  @doc false
  def changeset(post, attrs) do
    post
    |> cast(attrs, [:username, :body ])
    |> validate_required([:username, :body])
    |> validate_length(:body, min: 2, max: 250)
  end
end
```

3º Paso - Añadir funciones like y retweet

- En los contextos se guarda la API del modelo y toda la lógica de negocio.
- Repo es un módulo que nos permite trabajar con la base de datos abstrayéndose de detalles.

- A mayores de las CRUD necesitamos dar like y retweet.
- Versión simplificada de las funciones.

```
def inc_likes(%Post{id: id}) do
    from(post in Post, where: post.id == ^id, select: post)
    |> Repo.update_all(inc: [likes_count: 1])

end

def inc_reposts(%Post{id: id}) do
    from(post in Post, where: post.id == ^id, select: post)
    |> Repo.update_all(inc: [repost_count: 1])

end
```

Hablando de base de datos

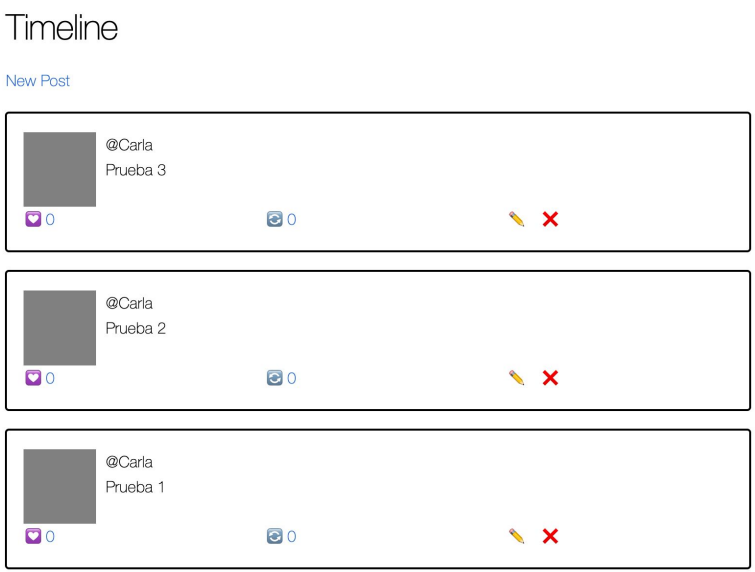
- Para ello, no trabajamos con scripts, SQL o noSQL.
- Para crear la tablas, alterarlas o borrarlas se usan migraciones.
- Localizadas en priv/repo/migrations.
- Gracias a Ecto tenemos un adapter para nuestro tipo de base de datos.

```
defmodule Demoatlant.Repo.Migrations.CreatePosts do
  use Ecto.Migration

  def change do
    create table(:posts) do
      add :username, :string
      add :body, :string
      add :likes_count, :integer
      add :repost_count, :integer

      timestamps()
    end
  end
end
```

docs: https://hexdocs.pm/ecto/Ecto.html

4º Paso - Crear nuestra vista

Nuestro objetivo visual



- Los estilos son fácilmente alcanzables con CSS o Tailwind.
- Debemos descomponer el timeline en posts.
- Todos los posts son iguales y deben comportarse igual.
- Deben tener comportamiento en tiempo real.

LiveView

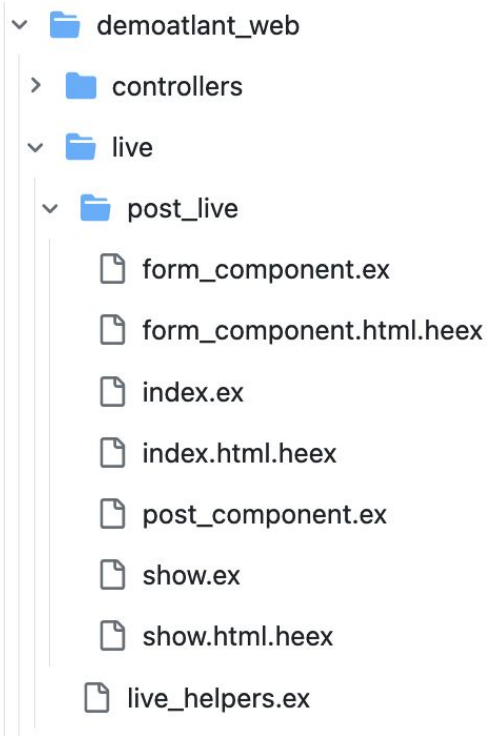
Aplicaciones SPA en Elixir sin javascript

LEARNING CONSTANTLY

We embrace new challenges and possibilities.

¿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.
- La comunicación servidor navegador va a través de un websocket, donde se guarda y manipula el estado.
- Los componentes de LiveView son: Liveview, LiveComponent y Component.




```
<div id="posts" >
  <%= for post <- @posts do %>

    <%= live_component DemoatlantWeb.PostLive.PostComponent, id: post.id, post: post %>
  <% end %>
</div>
```

4° Paso - Crear nuestra lista de tweets

```
defmodule DemoatlantWeb.PostLive.PostComponent do
  use DemoatlantWeb, :live_component
  import Demoatlant

  def render(assigns) do
    ~H"""
     <div  style="margin-top: 20px">
        <div id={"post-#{@post.id}"} style="outline: auto; padding: 20px" class="post">

          Código HTML vario, visitar Github para más ;D

        </div>
      </div>
    """
  end

  def handle_event("delete", _, socket) do
    {:noreply, socket}
  end

  def handle_event("like", _, socket) do
    Demoatlant.Timeline.inc_likes(socket.assigns.post)
    {:noreply, socket}
  end

  def handle_event("repost", _, socket) do
    Demoatlant.Timeline.inc_reposts(socket.assigns.post)
    {:noreply, socket}
  end

end
```

- Creamos un LiveComponent con lógica y código html en el mismo módulo.
- Encapsula toda la lógica como cualquier librería frontend de JS.
- Las funciones *handle_event/3* reaccionarán a eventos del usuario.
- El estado se almacenarán el los *assigns*.

5º Paso - Añadir comportamiento real time

- Phoenix LiveView emplea un esquema PubSub.
- Desde el modelo se exponen funciones para suscribirse a los eventos.
- Desde los métodos del modelo se llaman a las funciones que avisan a las vistas suscritas.

```
def subscribe do
  Phoenix.PubSub.subscribe(Demoatlant.PubSub, "posts")
end

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

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

```
    iex> update_post(post, %{field: bad_value})
    {:error, %Ecto.Changeset{}}

"""
def update_post(%Post{} = post, attrs) do
  post
  |> Post.changeset(attrs)
  |> Repo.update()
  |> broadcast(:post_updated)

end
```

5º Paso - Añadir comportamiento real time

- Desde la vista estos broadcast se reciben como mensajes de procesos normales de Elixir.
- *handle_info/2* recibe los mensajes y se ejecuta si matchea con la función.
- En estas funciones se actualiza el estado del socket y hace que este recargue la vista y se le envíe al cliente.

```
@impl true
 def handle_info({:post_created, post}, socket) do

   {:noreply, update(socket, :posts, fn posts -> [post | posts] end)}
 end

 @impl true
 def handle_info({:post_updated, _post}, socket) do

   {:noreply, update(socket, :posts, fn _posts -> list_posts() end)}
 end

 @impl true
 def handle_info({:post_deleted, _}, socket) do

   {:noreply,  update(socket, :posts, fn _posts -> list_posts() end )}
 end
```

¿Vídeo?¿Demo?

Por cierto, los tests....

Como poder mejorar este proyecto

- Crear una API para consumir -> mix phx.gen.json
- Crear una versión móvil nativa -> LiveView native

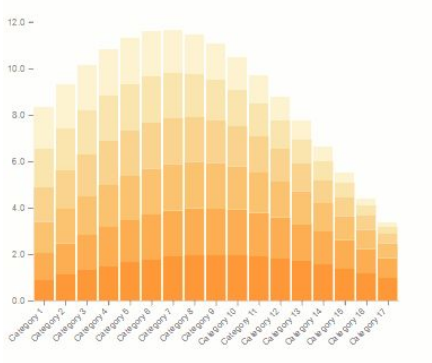
- Queremos continuar con librerías como React pero con nuestra app en Elixir ->

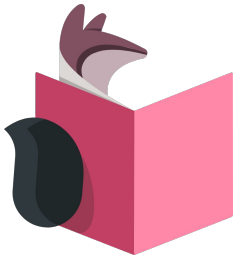
  https://blog.logrocket.com/to-do-list-phoenix-react-typescript/
- Añadir autenticación y autorización ->

  https://www.leanpanda.com/blog/authentication-and-authorisation-in-phoenix-liveview
- Mejorar IU -> Petal components, CSS, Tailwind.

Que más nos puede ofrecer Elixir

- Trabajar con redes neuronales en Elixir -> Axon (https://dockyard.com/blog/2022/01/11/getting-started-with-axon)
- Gráficos y data visualization aprovechando liveview-> Paquete context (https://github.com/mindok/contex)
- LiveBook -> Un proyecto muy nuevo que sirve para documentar, desplegar aplicaciones, visualizar datos, trabajar con modelos de machine learning.





Recomendaciones personales

Tutorial reconocimiento de dígitos

Image classification

Linkedin

Repositorio

Thank you