

Mi primera API REST con Phoenix y Postgres

Fecha de entrega: 13/septiembre.

Introducción

Para esta actividad vas a completar una guía de creación para terminar de generar una API con elixir y phoenix.

Tendrás a tu disposición las instrucciones a seguir con los comandos a ejecutar. Te corresponde seguirlos hasta llegar a la funcionalidad final.

Contexto

El objetivo principal es lograr una REST API como la demostrada en la sesión de código en vivo del día miércoles. Con la diferencia de que a cada uno le corresponderá un tema diferente el cual encontrarán a continuación en la siguiente tabla:

Aprendiz	Para un - Existen múltiples
ABDIAS EMMANUEL VERGARA RODRIGUEZ	Banco - Cliente
ALEJANDRO JOSE TORTOLERO MACHADO	Artista - Pintura
ALEX DAVID ESCUDERO SANCHEZ	Entrenador - Jugador
DAVID ELIAS CORREAL SANCHEZ	Universidad - Estudiante
EDWARD FABIAN OSORIO CAÑIZARES	Serie - Personaje
EDWIN MARINO MONTAÑO ANDRADE	Escritor - Libro
HERNANDO PARODI VARELA	Coreógrafo - Bailarines
JEISON DANIEL MORENO MORENO	Sello Discográfico - Cantante
JORGE IVAN ROJAS GUERRA	País - Ciudades
JORGE MARIO ARANGO GRISALES	Doctor - Paciente

JOSE DANIEL RESTREPO HENAO	Zoológico - Animal
JOSE ISAURO JAIME MIRANDA	Técnico - Empresa
JUAN DAVID MENDEZ CAMPUZANO	Noticiero - Presentador
LEIDY JOHANA RESTREPO MONTOYA	Restaurante - Platos
LUIS FERNANDO ARCINIEGAS RIVERA	Persona - Propiedad (inmuebles)
MONICA ALEXANDRA CARRILLO HERRERA	Repartidor - Encomienda
MARIA ISABEL RESTREPO AGUDELO	Banda musical - Integrante
NATALIA ANDREA VERA TORRES	Programador - Tareas
WENDY JULIETH RUEDA ESTUPIÑAN	Planetas - Lunas

Tener en cuenta que estas relaciones deben limitarse a funcionar de la siguiente manera: Para un X hay múltiples Y's.

Por hacer

Generar proyecto

Reemplazar lo resaltado por el nombre de tu aplicación - 0.5p

```
>> mix phx.new app_name --no-install --app app_name --database
postgres --no-live --no-assets --no-html --no-dashboard --no-mailer
--binary-id
```

Configuración de PostgreSQL

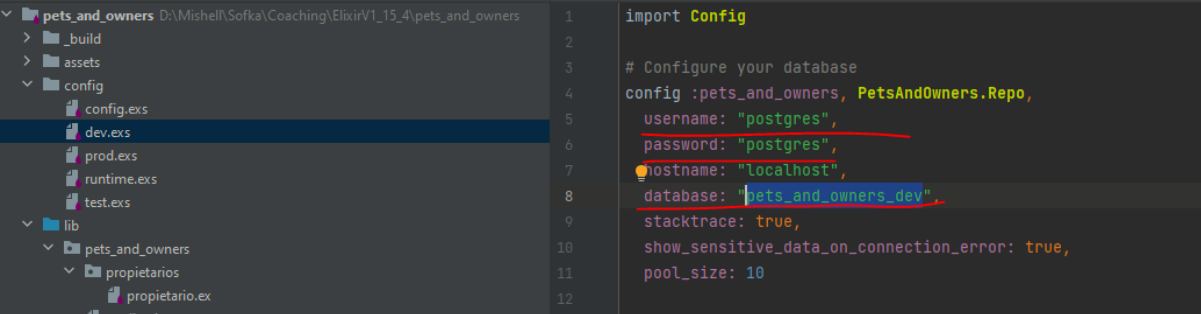
Dos opciones: Instalar postgres normal o por medio de docker.

Para docker el comando es el siguiente.

```
>> docker run --name app_name_bd -p 5432:5432 -e
POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -d postgres_img
```

Se generará el contenedor con el nombre designado en lo resaltado, el cual debe ser editado en el siguiente archivo dentro de sus proyectos de phoenix.

Así mismo, si Postgres es local, debes hacer que los nombres sean idénticos así como el user y password.

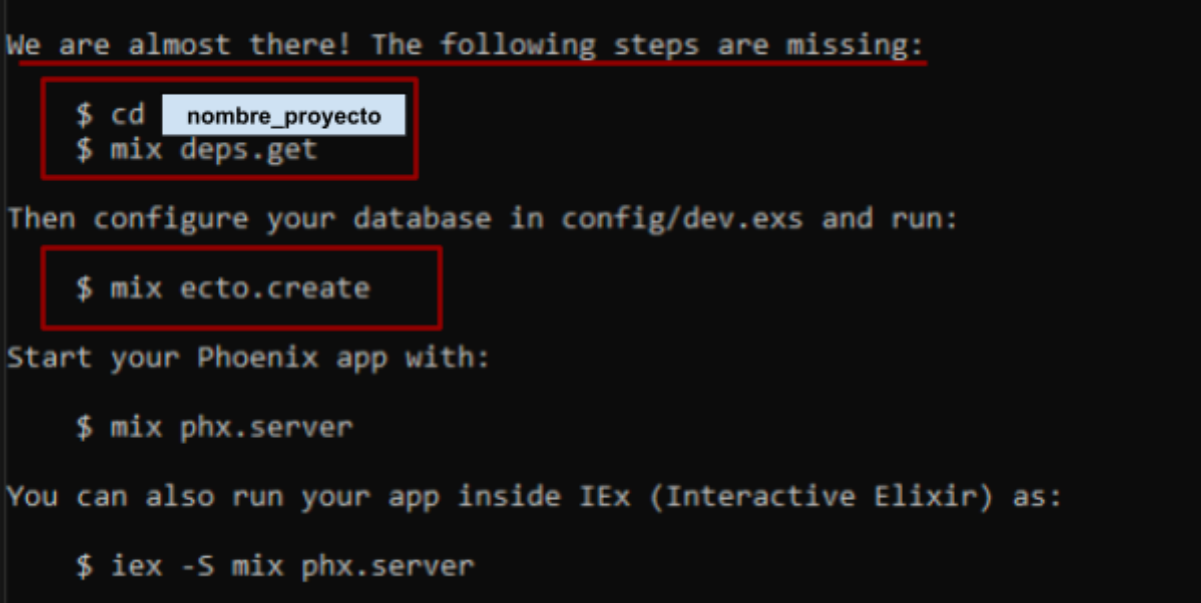


```
1 import Config
2
3 # Configure your database
4 config :pets_and_owners, PetsAndOwners.Repo,
5   username: "postgres",
6   password: "postgres",
7   hostname: "localhost",
8   database: "pets_and_owners_dev",
9   stacktrace: true,
10  show_sensitive_data_on_connection_error: true,
11  pool_size: 10
12
```

No hay puntos en este ítem.

Ejecuciones primarias

En este punto la base de datos debe estar configurada, para ejecutar los siguientes comandos - 0.5p



```
We are almost there! The following steps are missing:

$ cd nombre_proyecto
$ mix deps.get

Then configure your database in config/dev.exs and run:

$ mix ecto.create

Start your Phoenix app with:

$ mix phx.server

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

$ iex -S mix phx.server
```

Creación de entidades

Vas a generar las dos entidades de acuerdo al tema asignado, recuerde la importancia de cómo deben enviarse los nombres para la generación de la entidad, el servicio y el controlador. - 1p

```
>> mix phx.gen.json Entidades1 Entidad1 entidades1 campo1:tipo,  
campo2:tipo ...
```

La entidad que lleva la clave foránea

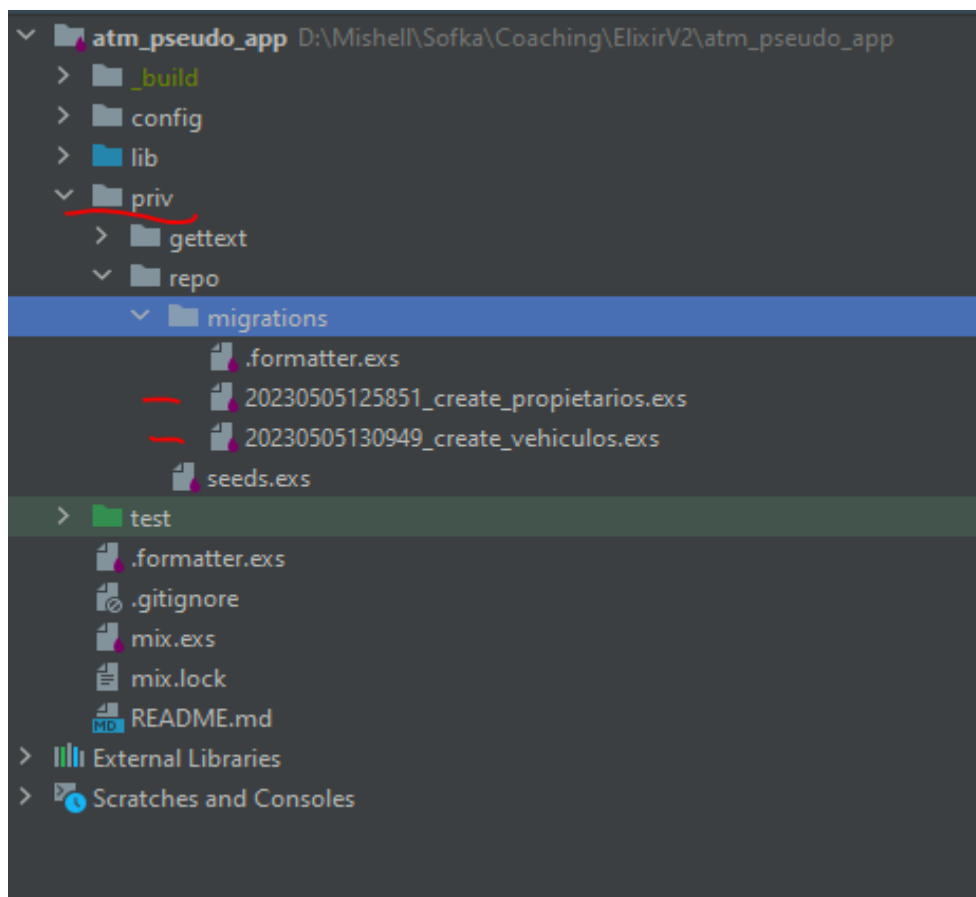
```
>> mix phx.gen.json Entidades2 Entidad2 entidades2  
entidad1_id:references:entidades1 campo1:tipo, campo2:tipo ...
```

Ejemplo:

```
mix phx.gen.json Propietarios Propietario propietarios  
num_iden:string nombres:string apellidos:string fecha_nac:date  
edad:integer, ...
```

```
mix phx.gen.json Vehiculos Vehiculo vehiculos  
propietario_id:references:propietarios placa:string tipo:string  
modelo:string marca:string anio:integer color:string
```

Verificar que los archivos de migración fueron correctamente generados en la siguiente carpeta. Como se van a generar dos entidades, luego de la ejecución de los comando anteriores, debería haber dos archivos en este directorio.



Migración de las tablas

En esta sección vas a mandar a generar las tablas pero antes, debemos hacer los cambios para que la relación de nuestras tablas sea cargada a PostgreSQL

Buscamos el archivo de migración donde seteamos lo de la clave foránea en los comandos de la sección anterior y haces que la parte señalada en la línea a continuación, se parezca a la de tu código. - 0.5p

```
1 defmodule AtmPseudoApp.Repo.Migrations.CreateVehiculos do
2   use Ecto.Migration
3
4   def change do
5     create table(:vehiculos, primary_key: false) do
6       add :id, :binary_id, primary_key: true
7       add :placa, :string
8       add :tipo, :string
9       add :marca, :string
10      add :modelo, :string
11      add :color, :string
12      add :anio, :integer
13      add :propietario_id, references(:propietarios, on_delete: :delete_all, type: :binary_id)
14
15      timestamps()
16    end
17
18    create index(:vehiculos, [:propietario_id, :placa])
19  end
20 end
21
```

Si tiene problemas con la migración, recuerda que si una entidad es dependiente de otra, esta es la que debe migrar primero. Es decir, por su nombre Ecto sabrá quién va primero.

Ahora es momento de decirle a Elixir también que la relación existe. Vamos a los archivos representantes de las entidades (Entidad1.ex y Entidad2.ex, nombre en singular) y colocamos correctamente las propiedades de `has_many` y `belongs_to`. - 0.5p

Recuerde:

belongs_to va en la entidad cuyo archivo de migración tenga la configuración de la captura anterior. Por lo tanto, esto implica automáticamente que *has_many* va en la otra entidad.

```

1 defmodule AtmPseudoApp.Vehiculos.Vehiculo do
2   use Ecto.Schema
3   import Ecto.Changeset
4
5   @primary_key {:id, :binary_id, autogenerate: true}
6   @foreign_key_type :binary_id
7   schema "vehiculos" do
8     field :anio, :integer
9     field :color, :string
10    field :marca, :string
11    field :modelo, :string
12    field :placa, :string
13    field :tipo, :string
14    belongs_to :propietario, AtmPseudoApp.Propietarios.Propietario
15    timestamps()
16  end
17
18  @doc false
19  def changeset(vehiculo, attrs) do
20    vehiculo
21    |> cast(attrs, [:placa, :tipo, :marca, :modelo, :color, :anio, :propietario_id])
22    |> validate_required([:placa, :tipo, :marca, :modelo, :color, :anio])
23    |> validate_format(placa, ~r/^[a-zA-Z]{3}-\d{4}$|^[a-zA-Z]{2}-\d{3}[a-zA-Z]$/, message: "Invalid placa")
24    |> validate_length(placa, max: 8, message: "Placa must have up to 8 characters")
25    |> unique_constraint(:placa, message: "Placa already exists")
26  end
27 end
28

```

```

1 defmodule AtmPseudoApp.Propietarios.Propietario do
2   use Ecto.Schema
3   import Ecto.Changeset
4
5   @primary_key {:id, :binary_id, autogenerate: true}
6   @foreign_key_type :binary_id
7   schema "propietarios" do
8     field :apellidos, :string
9     field :fecha_nac, :date
10    field :nombres, :string
11    field :num_id, :string
12    has_many :vehiculos, AtmPseudoApp.Vehiculos.Vehiculo
13    timestamps()
14  end
15
16  @doc false
17  def changeset(proprietario, attrs) do
18    propietario
19    |> cast(attrs, [:num_id, :nombres, :apellidos, :fecha_nac])
20    |> validate_required([:num_id, :nombres, :apellidos, :fecha_nac])
21    |> validate_length(:num_id, max: 10, message: "Identification (num_id) must have up to 10 digits")
22    |> unique_constraint(:num_id, message: "Proprietario already exists with this identification (num_id)")
23  end
24 end
25
26

```

Finalmente, puedes ejecutar el comando

```
>> mix ecto.migrate
```

Configuración para vistas

Ahora vamos a hacer los últimos ajustes para que nuestra relación cargue al momento de realizar las debidas consultas

1. Agregar los respectivos `Repo.preload(:entidad1)` y `Repo.preload(:entidad2)` en los servicios de ambas entidades. - 0.5p

Ejemplos:

```
def list_propietarios do
  Repo.all(Propietario) |> Repo.preload(:vehiculos)
end

@doc
```

```
def get_propietario!(id), do: Repo.get!(Propietario, id) |> Repo.preload(:vehiculos)
```

```
def create_propietario(attrs \\ %{}) do
  %Propietario{}
  |> Repo.preload(:vehiculos)
  |> Propietario.changeset(attrs)

  |> Repo.insert()
end
```

```
def list_vehiculos do
  Repo.all(Vehiculo) |> Repo.preload(:propietario)
end
```

```
def get_vehiculo!(id), do: Repo.get!(Vehiculo, id) |> Repo.preload(:propietario)
```

```

def create_vehiculo(attrs \\ %{}) do
  %Vehiculo{}
  |> Repo.preload(:propietario)
  |> Vehiculo.changeset(attrs)
  |> Repo.insert()
end

```

Precaución: El átomo que utilices es el referente configurado en la vista que harás para que esto sea visible.

2. Configurar los archivos de las vistas

Debes ir a cada archivo de las vistas (entidad1_json.ex y entidad2_json.ex) y configurar para que se vean las relaciones. - 0.5p

Por ejemplo

```

defp data(%Propietario{} = propietario) do
  %{
    id: propietario.id,
    num_id: propietario.num_id,
    nombre: propietario.nombre,
    apellido: propietario.apellido,
    fecha_nac: propietario.fecha_nac,
    vehiculos: for(v <- propietario.vehiculos, do: data_vehiculos(v)) #Many vehiculos - TODO
  }
end

#Structure to load vehiculos
defp data_vehiculos(%Vehiculo{} = vehiculo) do
  %{
    id: vehiculo.id,
    anio: vehiculo.anio,
    color: vehiculo.color,
    marca: vehiculo.marca,
    modelo: vehiculo.modelo,
    placa: vehiculo.placa,
    tipo: vehiculo.tipo
  }
end

```

Handwritten red annotations: A red arrow points to the `vehiculos` field in the first struct. A large red bracket and the word "TODO" are next to the `data_vehiculos` function definition.


```

defp data(%Vehiculo{} = vehiculo) do
  %{
    id: vehiculo.id,
    placa: vehiculo.placa,
    tipo: vehiculo.tipo,
    marca: vehiculo.marca,
    modelo: vehiculo.modelo,
    color: vehiculo.color,
    anio: vehiculo.anio,
    propietario: data_propietario(vehiculo.propietario) #A single propietario
  }
end

#Structure to load propietarios
defp data_propietario(proprietario) when is_nil(proprietario), do: :ok

defp data_propietario(%Propietario{} = propietario) do
  %{
    id: propietario.id,
    num_id: propietario.num_id,
    nombre: propietario.nombre,
    apellido: propietario.apellido,
    fecha_nac: propietario.fecha_nac
  }
end

```

Handwritten red annotations: A red arrow points to the `propietario` field in the first struct. A red circle with "TODO" is next to the `#A single propietario` comment. Another red circle with "TODO" is next to the `data_propietario` function call in the second struct.

Configurar las rutas

Para ir finalizando, vamos a configurar las rutas, en el archivo `router.ex` - 0.5p

```

resources "/propietarios", PropietarioController, except: [:new, :edit, :update]
put "/propietarios/:id", PropietarioController, :update

resources "/vehiculos", VehiculoController, except: [:new, :edit, :update]
put "/vehiculos/:id", VehiculoController, :update

```

Lo tachado en rojo no es necesario.

Ejecución del proyecto

Para este punto, debes levantar la API y mostrar funcionalidad de la siguiente manera:

1. Ingresar información de acuerdo a su tema - 2.5p
2. Realizar consultas a cada entidad de todos los registros - 2p
3. Realizar consultas a cada entidad de un registro por el id. - 1p

Comando para ejecutar un proyecto en phoenix

```
>> mix phx.server
```

Entregables

- Video de no más de 5 minutos que debe mostrar la sección de Ejecución del proyecto.
- Enlace a repositorio github del proyecto.
- Proyectos entregados luego de la fecha dada, serán recibidos hasta el 15 de septiembre con máxima calificación a recibir de 7,5/10.