Elixir

Programación más funcional que nunca

Semana 3 – MC #5

SofkaU

#ElDesafíoEsContigo

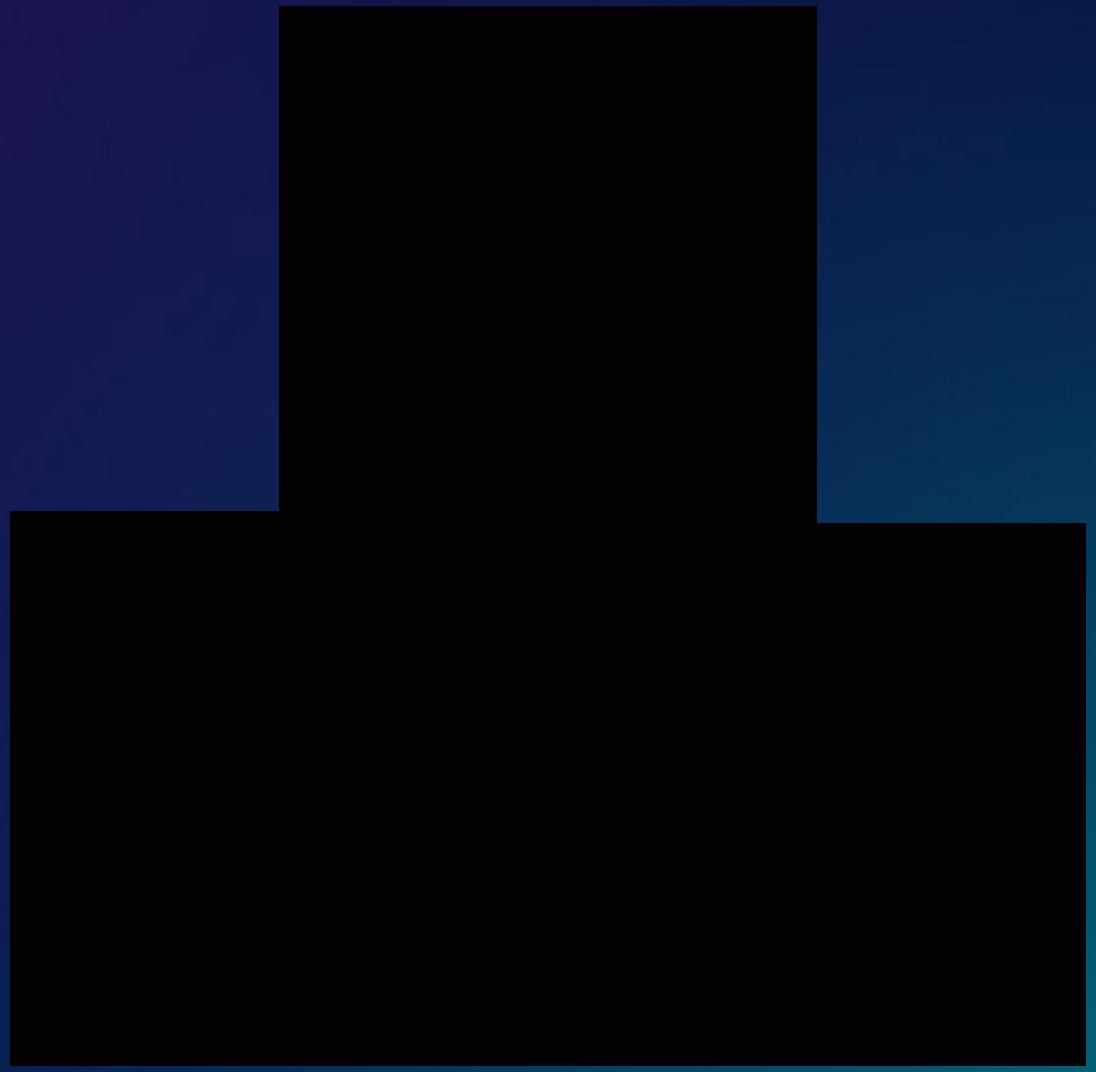
Temas

01 Procesos

02 Genserver

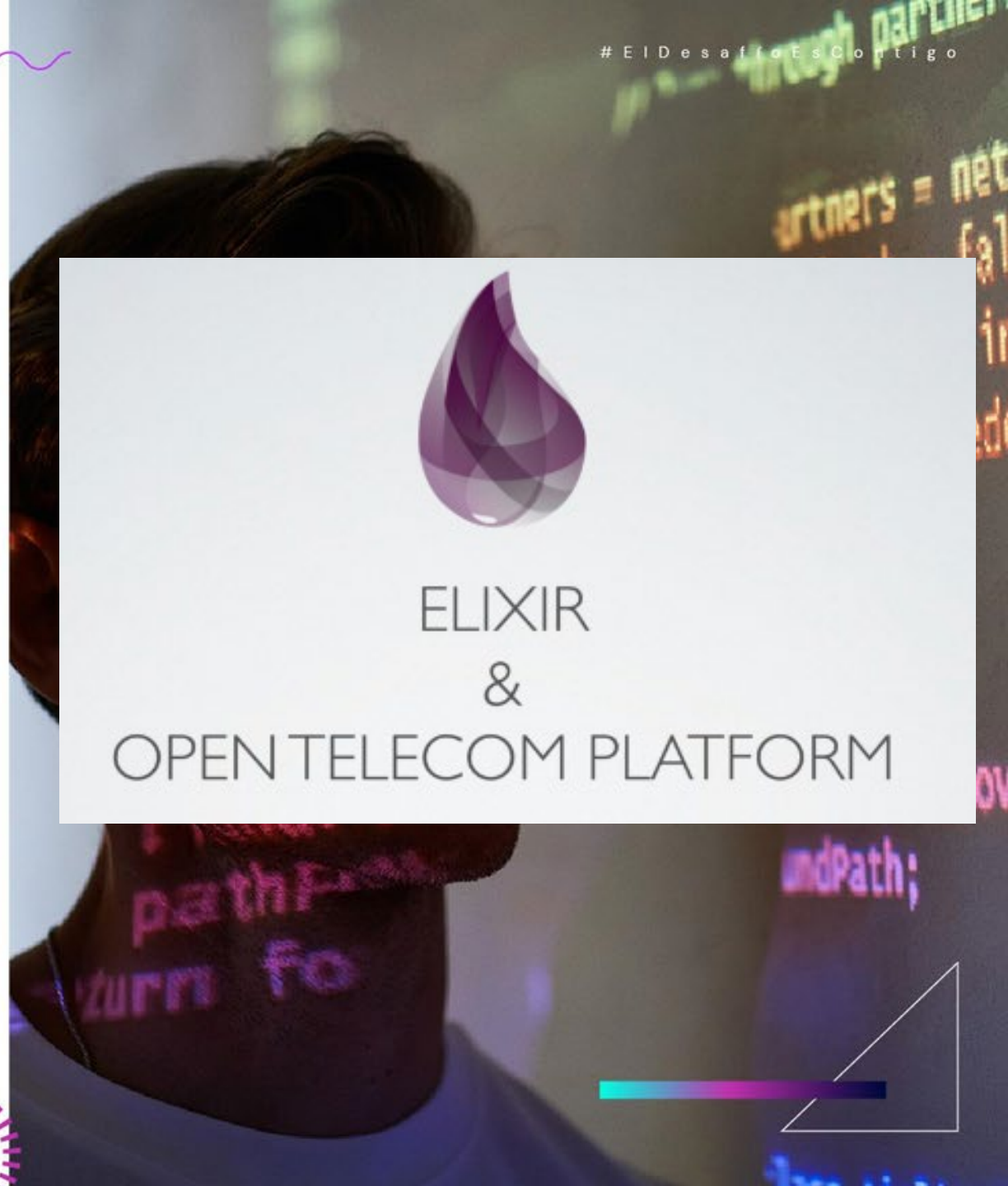
03 Preguntas y respuestas

Elixir
Procesos



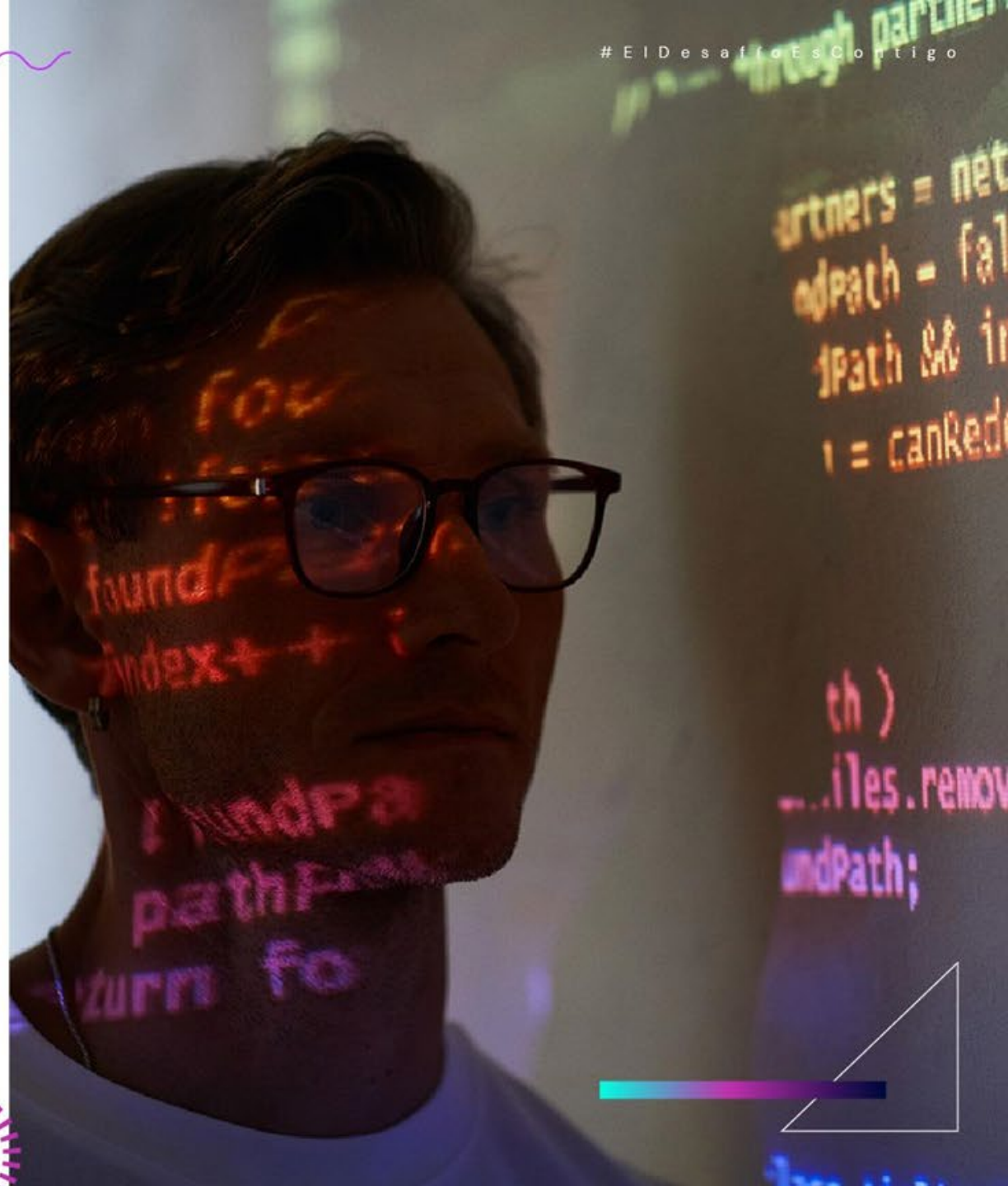
Open Telecom Platform

- Basado en Erlang
- Contiene un enorme conjunto de bibliotecas de BEAM que siguen los principios de diseño de sistemas.
- Cuando hablamos de OTP en el contexto de Elixir, normalmente nos referimos al modelo de actor.



Procesos

- Contexto de ejecución ligero que se ejecuta simultáneamente con otros procesos.
- Elixir también proporciona una serie de abstracciones construidas sobre los procesos, como tareas y agentes, que simplifican los casos de uso comunes.
- Se crean directamente en BEAM y se comunican de forma asíncrona mediante mensajes.

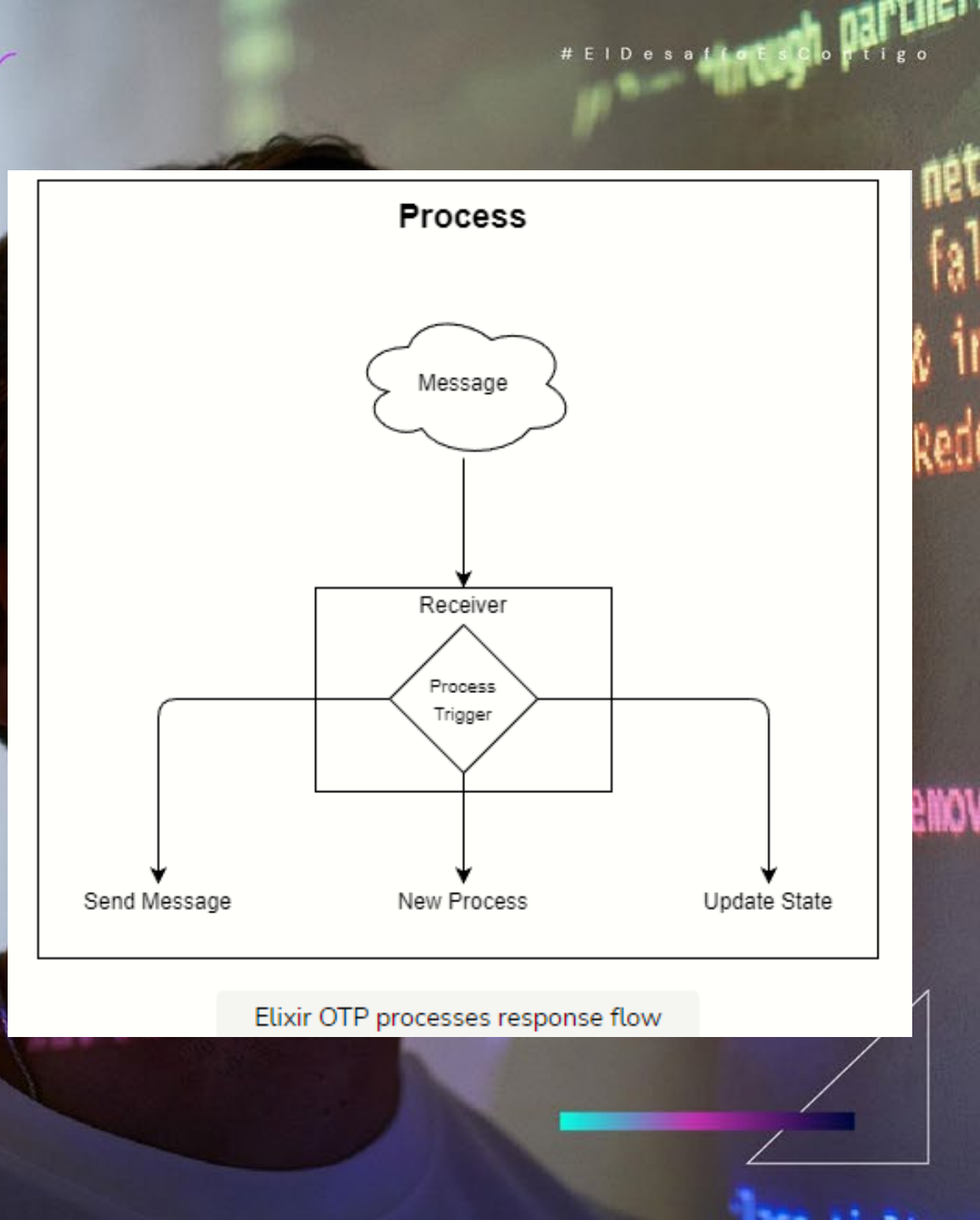


Procesos

Cada proceso recibe un mensaje. En respuesta a ese mensaje puede:

- Crear un nuevo proceso.
- Enviar un mensaje a otro proceso.
- Modificar su estado.

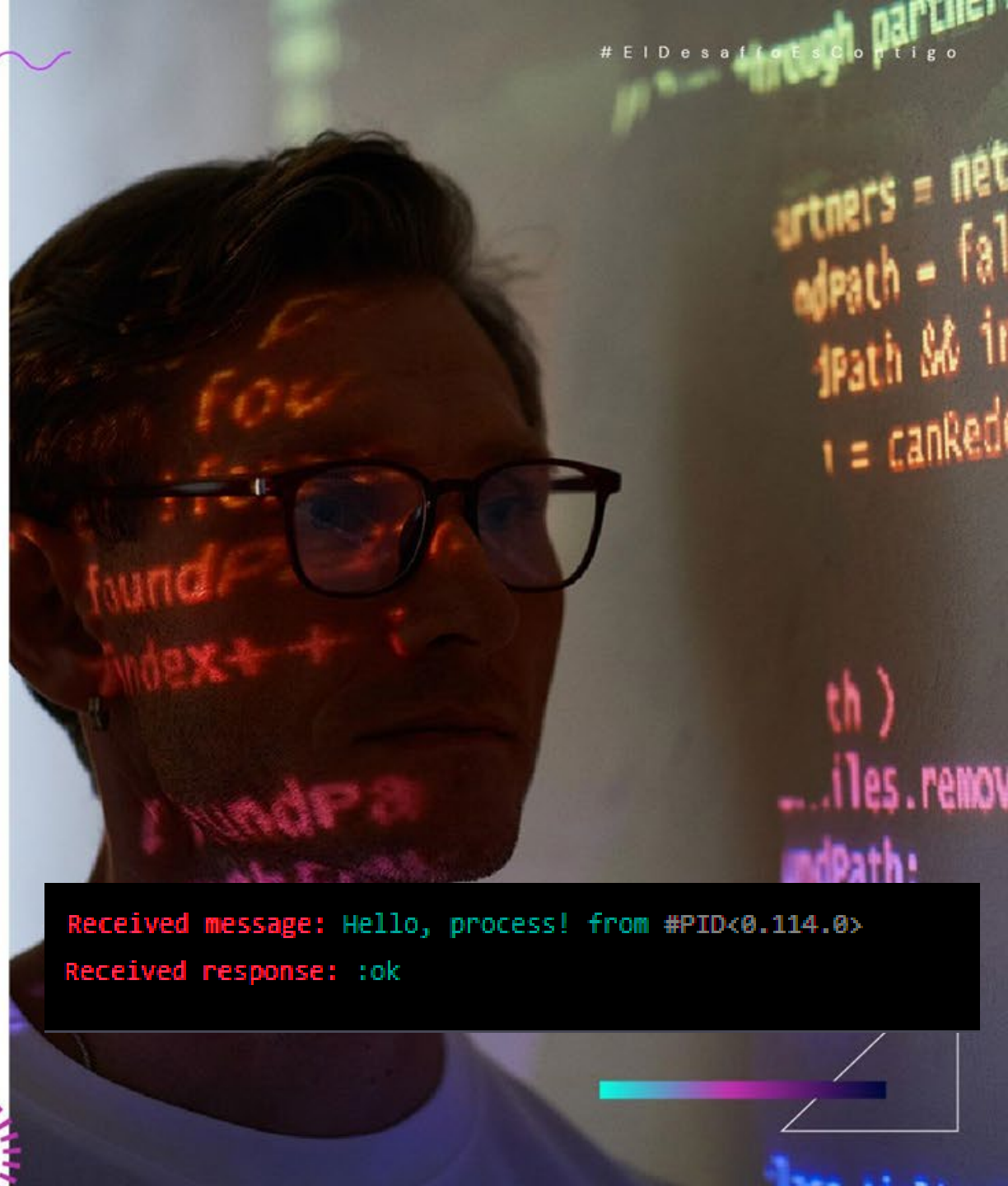
Los procesos siguen distintos patrones y, para controlarlos, utilizamos comportamientos.



Elixir OTP processes response flow

Ejemplo

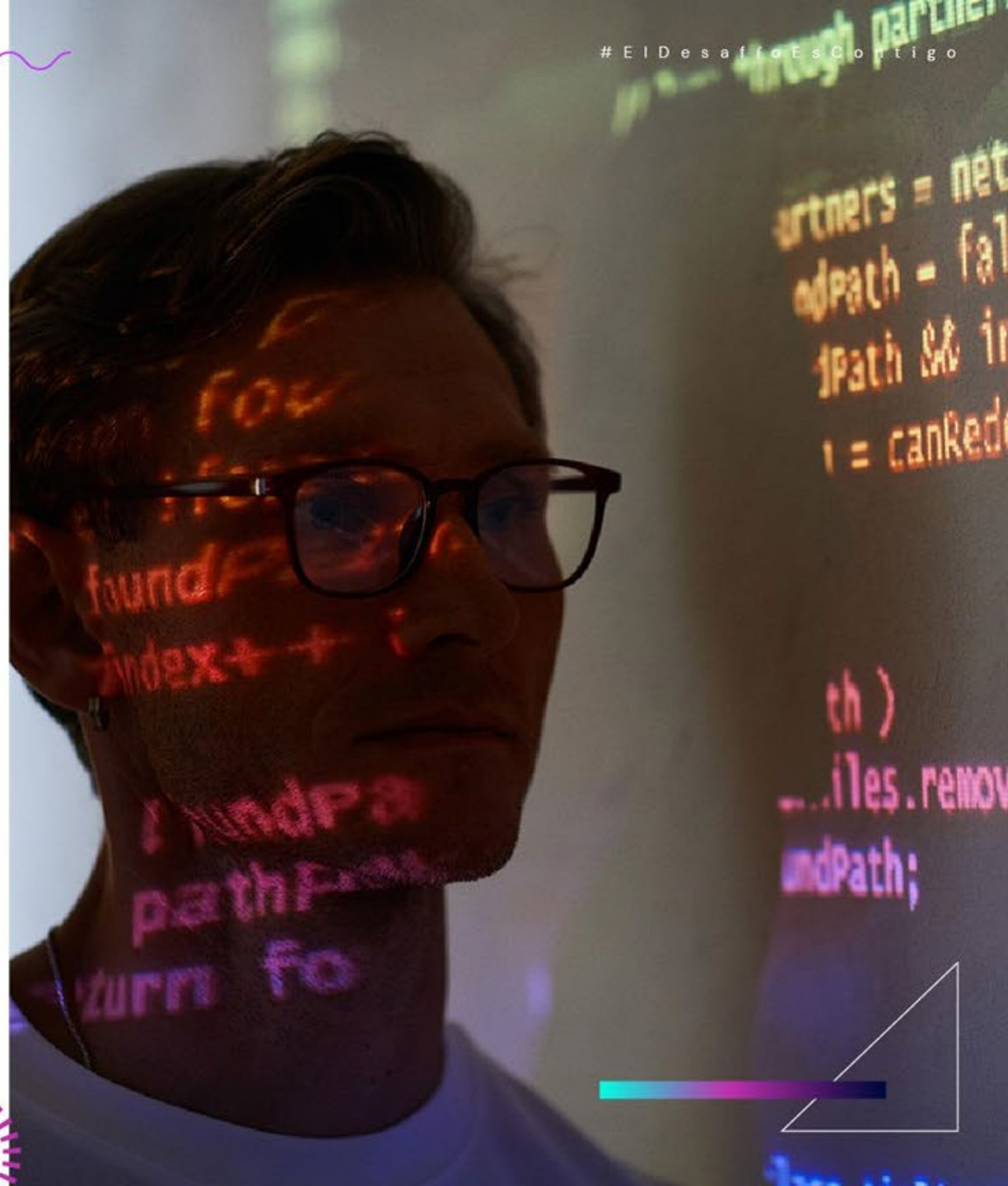
```elixir
# Define a function that will run in the new process
defmodule MyProcess do
  def run do
    receive do
      {sender_pid, message} ->
        IO.puts "Received message: #{message} from #{inspect sender_pid}"
    end
  end
end

# Spawn a new process and send it a message
pid = spawn(MyProcess, :run, [])

send(pid, {self(), "Hello, process!"})

# Wait for a response from the process
receive do
  message ->
    IO.puts "Received response: #{message}"
after
  5000 ->
    IO.puts "No response received"
end
```



Elixir Genserver

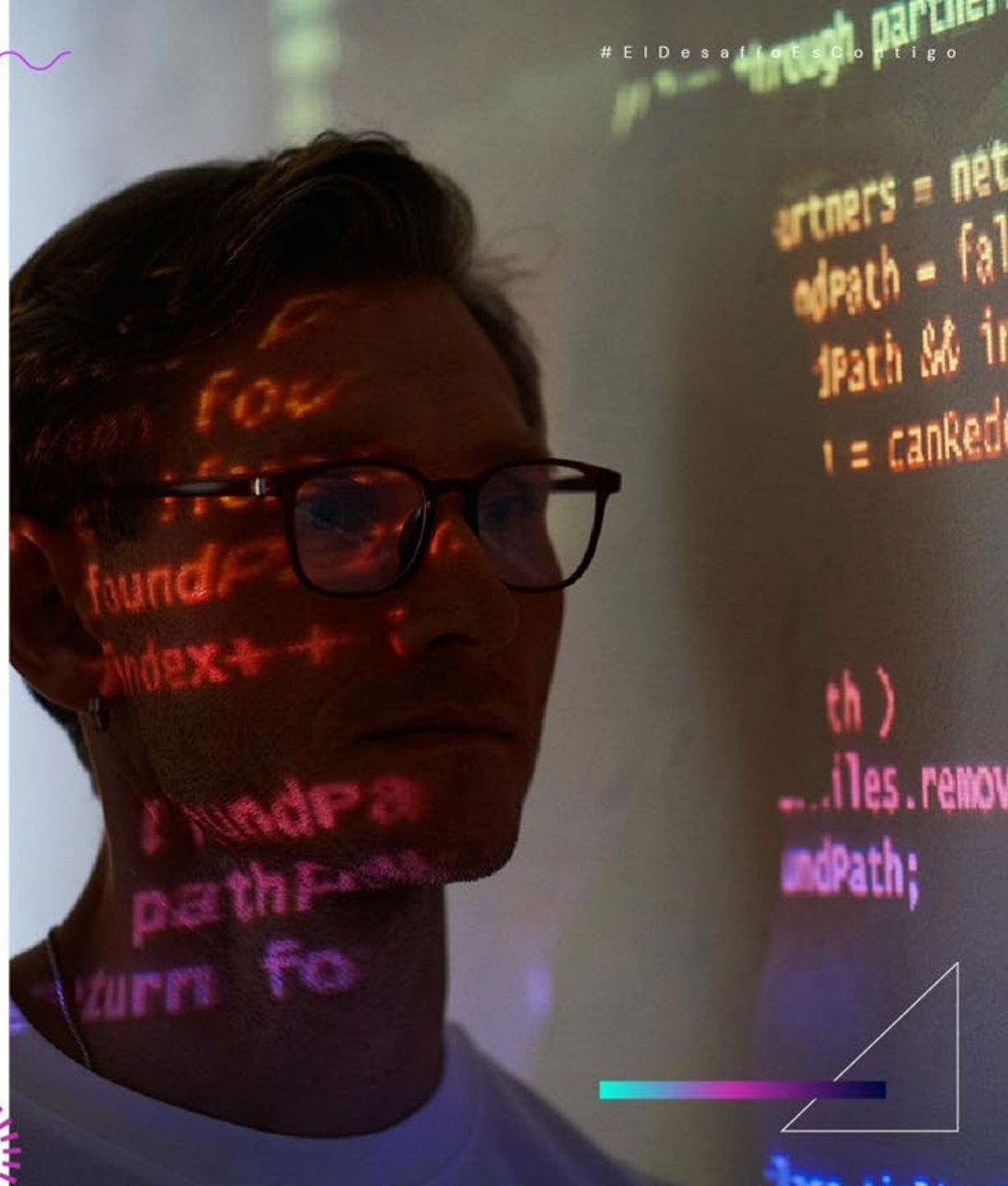
Servidor genérico

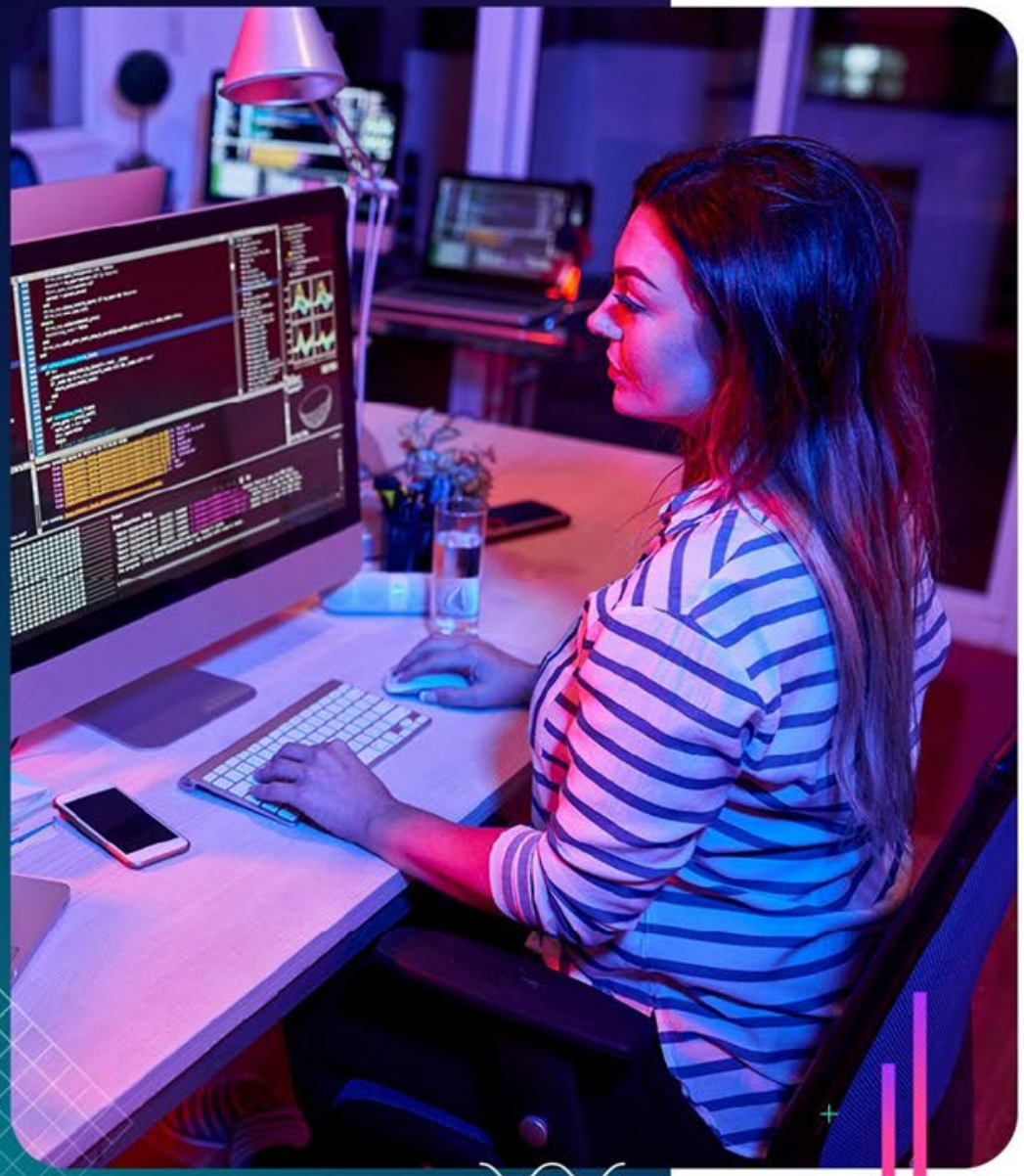
- GenServer es un módulo de comportamiento que se utiliza para implementar el servidor de una relación cliente-servidor.
- Es un proceso para mantener estados, ejecutar código de forma asíncrona, etc.
- Viene con el conjunto estándar de funciones para introducir funcionalidades tanto de seguimiento de errores como de informes.
- Puede formar parte de la supervisión.



Use

- Se utiliza habitualmente en Elixir para añadir funcionalidad a un módulo, "mezclando" código de otro módulo.
- Al utilizarlo, Elixir evalúa el módulo especificado como argumento para use, y ejecuta cualquier código que esté definido en ese módulo.





Uso de genserver

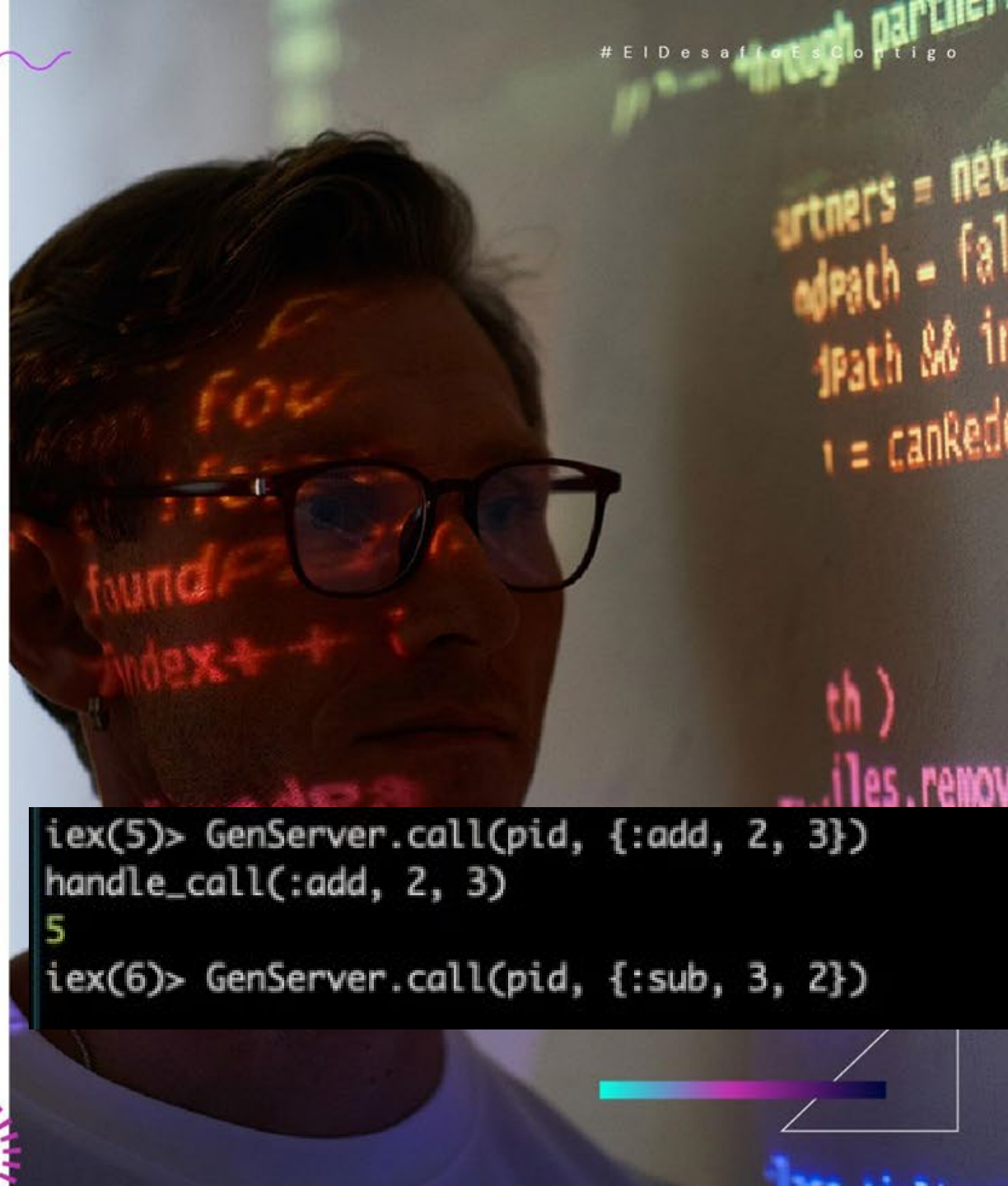
```
gcalc.exs
 7 defmodule GCalc do
 6   use GenServer
 5
 4   def init(param) do
 3     IO.puts "Inicio GenServer GCalc"
 2     IO.inspect param
 1     {:ok, %{conteo: 0}}
 8   end
 1 end
```

```
Interactive Elixir (1.11.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c("gcalc.exs")
[GCalc]
iex(2)> GenServer.start
start/2         start/3         start_link/2    start_link/3

iex(2)> GenServer.start_link(GCalc, {"hola", "buenas"})
Inicio GenServer GCalc
{"hola", "buenas"}
{:ok, #PID<0.115.0>}
```

Genserver

- handle_call/3 (msg, from, state): Lo más cercano para una dinámica Cliente-Servidor.

```elixir
defmodule GCalc do
  use GenServer

  def init(param) do
    IO.puts "Inicio GenServer GCalc"
    IO.inspect param
    {:ok, %{conteo: 0}}
  end

  def handle_call({:add, n, m}, _from, %{conteo: conteo}) do
    IO.puts "handle_call(:add, #{n}, #{m})"
    resul = n + m
    {:reply, resul, %{conteo: conteo + 1}}
  end

  def handle_info(:hello, %{conteo: conteo}) do
    IO.puts "He sido llamada #{conteo} veces"
    {:noreply, %{conteo: conteo + 1}}
  end

  def handle_info(:world, state) do
    {:noreply, state}
  end

end
```

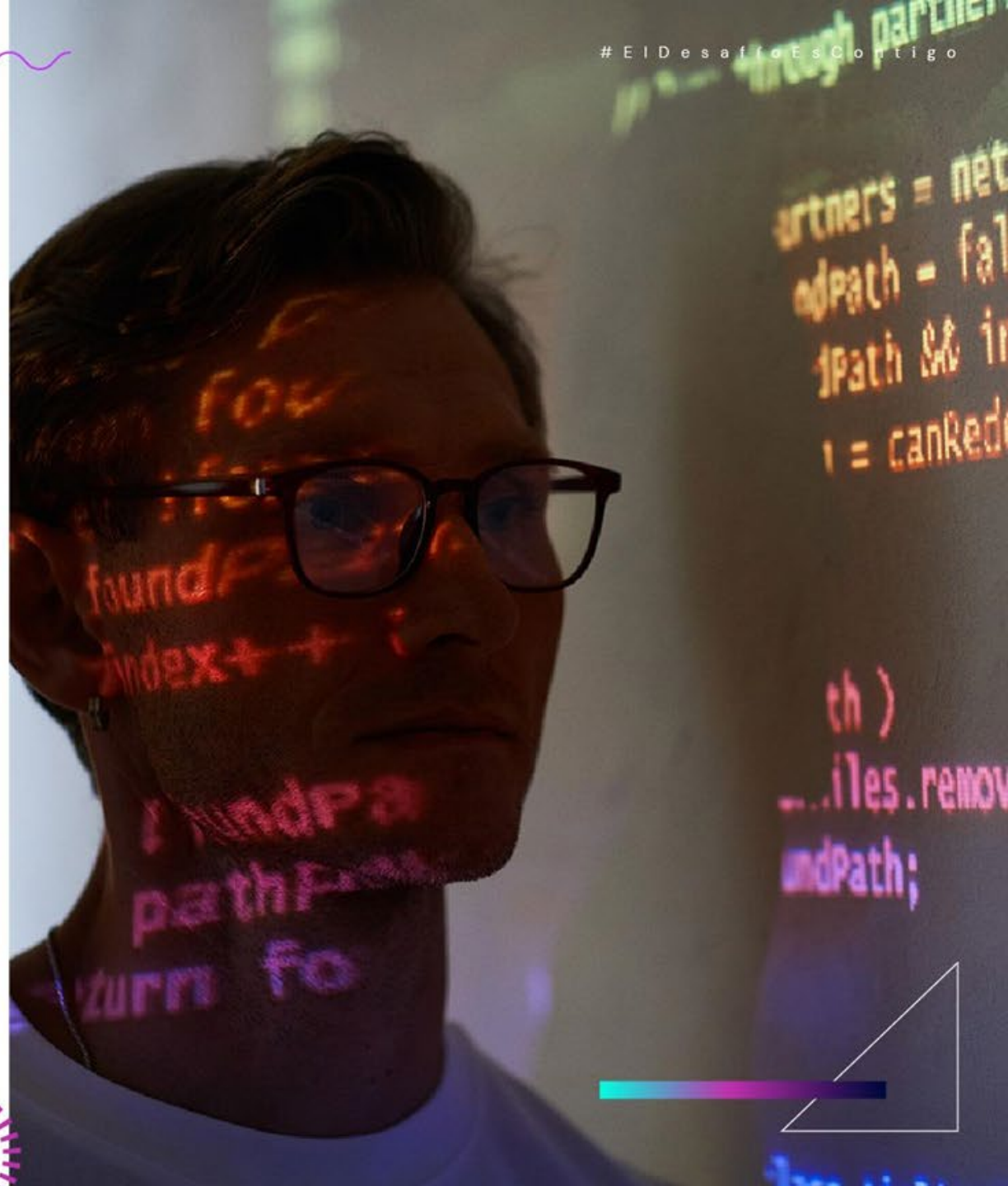


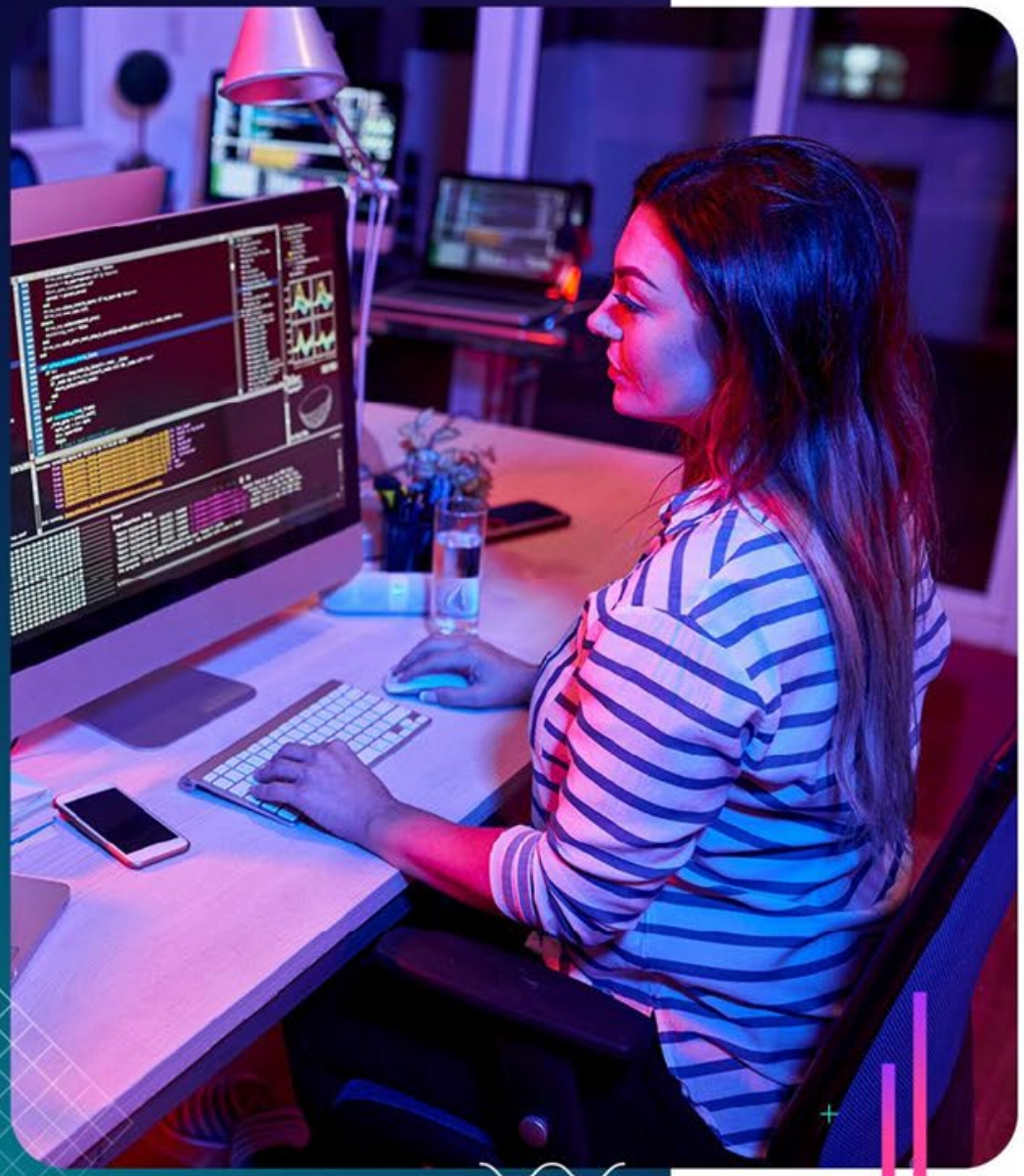
Genserver

- handle_call/3 (msg, from, state): Lo más cercano para una dinámica Cliente-Servidor.

```
def handle_call({:add, n, m}, _from, %{conteo: conteo}) do
  IO.puts "handle_call(:add, #{n}, #{m})"
  Process.sleep(2000)
  IO.puts "WAKEUP"
  resul = n + m
  {:reply, resul, %{conteo: conteo + 1}}
end
```

```
iex(2)> {:ok, pid} = GenServer.start(GCalc, [])
Inicio GenServer GCalc
[]
{:ok, #PID<0.141.0>}
iex(3)> GenServer.call(pid, {:add, 2, 3})
handle_call(:add, 2, 3)
WAKEUP
5
```





Uso de genserver

```
def handle_call({:add, n, m}, _from, %{conteo: conteo}) do
  IO.puts "handle_call(:add, #{n}, #{m})"
  Process.sleep(6000)
  IO.puts "WAKEUP"
  resul = n + m
  {:reply, resul, %{conteo: conteo + 1}}
end
```

```
iex(6)> GenServer.call(pid, {:add, 2, 3})
handle_call(:add, 2, 3)
** (exit) exited in: GenServer.call(#PID<0.141.0>, {:add, 2, 3}, 5000)
    ** (EXIT) time out
    (elixir 1.11.3) lib/gen_server.ex:1027: GenServer.call/3
WAKEUP
```

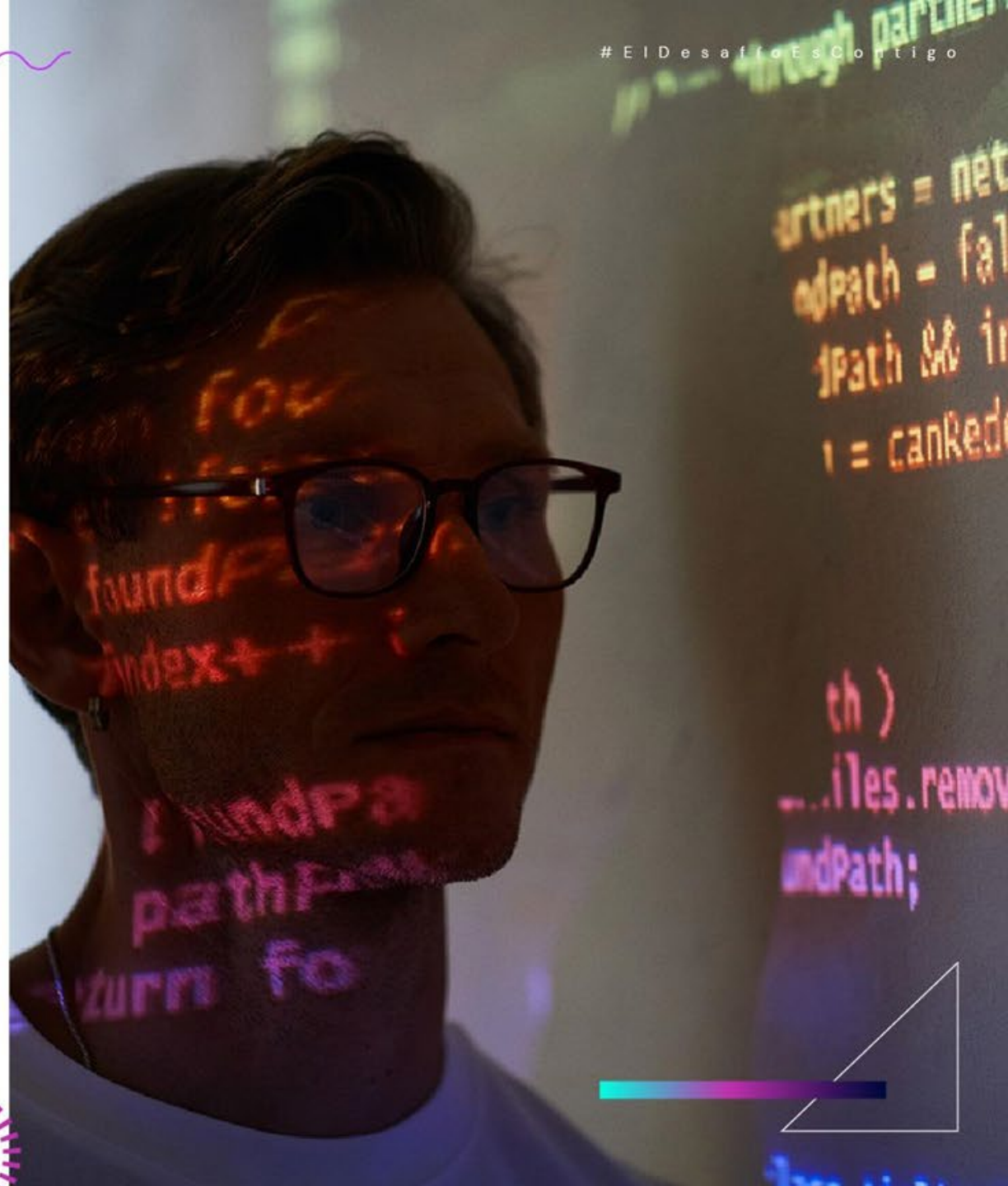
```
iex(8)> GenServer.call(pid, {:add, 2, 3}, 10000)
handle_call(:add, 2, 3)
WAKEUP
5
```

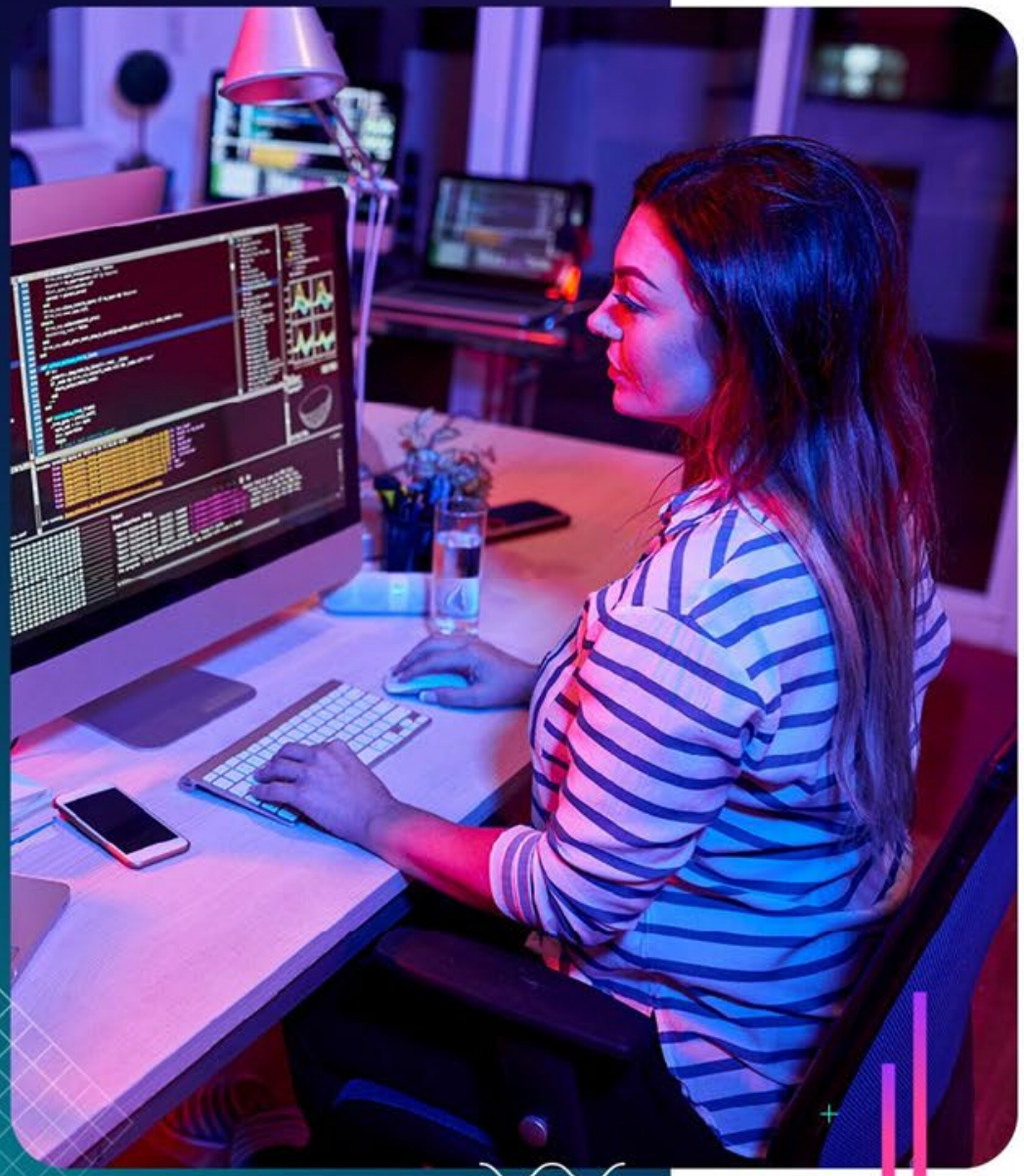
Genserver

- handle_cast/2 (msg, state): Cuando enviamos un mensaje pero no esperamos respuesta.

```
def handle_cast(:reset, _state) do
  IO.puts "handle_cast(:reset)"
  {:noreply, %{conteo: 0}}
end
```

```
iex(11)> r(GCalc)
warning: redefining module GCalc (current version defined in memory)
  gcalc.exs:1

{:reloaded, GCalc, [GCalc]}
iex(12)> GenServer.cast(pid, :reset)
handle_cast(:reset)
:ok
```





Envolventes

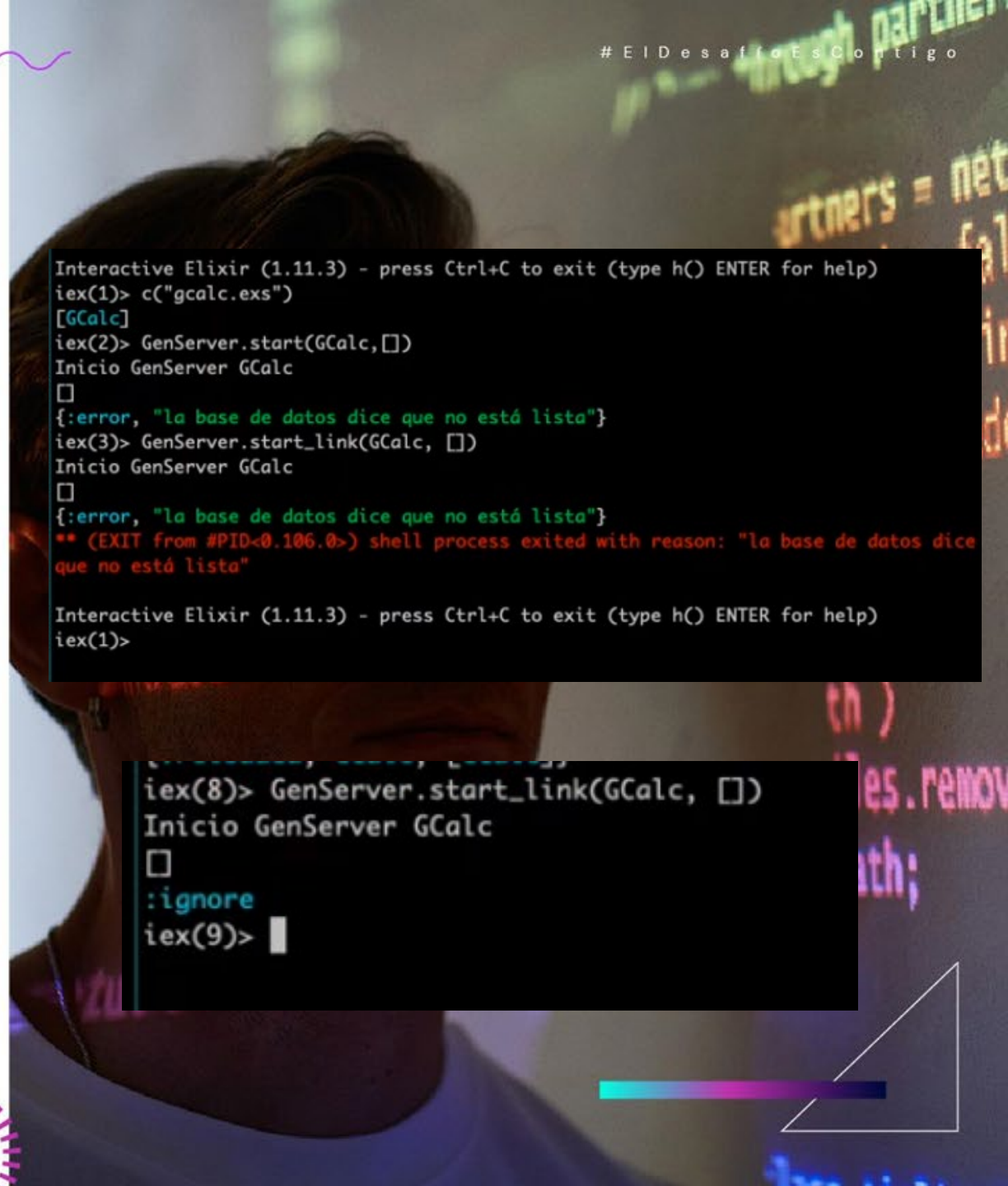
```elixir
def add(pid, x, y) do
  GenServer.call(pid, {:add, x, y})
end
```

```
iex(16)> r(GCalc)
warning: redefining module GCalc (current version defined in memory)
  gcalc.exs:1

{:reloaded, GCalc, [GCalc]}
iex(17)> {:ok, pid} = GenServer.start(GCalc, [])
Inicio GenServer GCalc
[]
{:ok, #PID<0.185.0>}
iex(18)> GCalc.add(pid, 2, 3)
handle_call(:add, 2, 3)
5
```

Control de errores

```elixir
def init(param) do
  IO.puts "Inicio GenServer GCalc"
  IO.inspect param
  # {:ok, %{conteo: 0}}

  {:stop, "la base de datos dice que no está lista"}
end
```

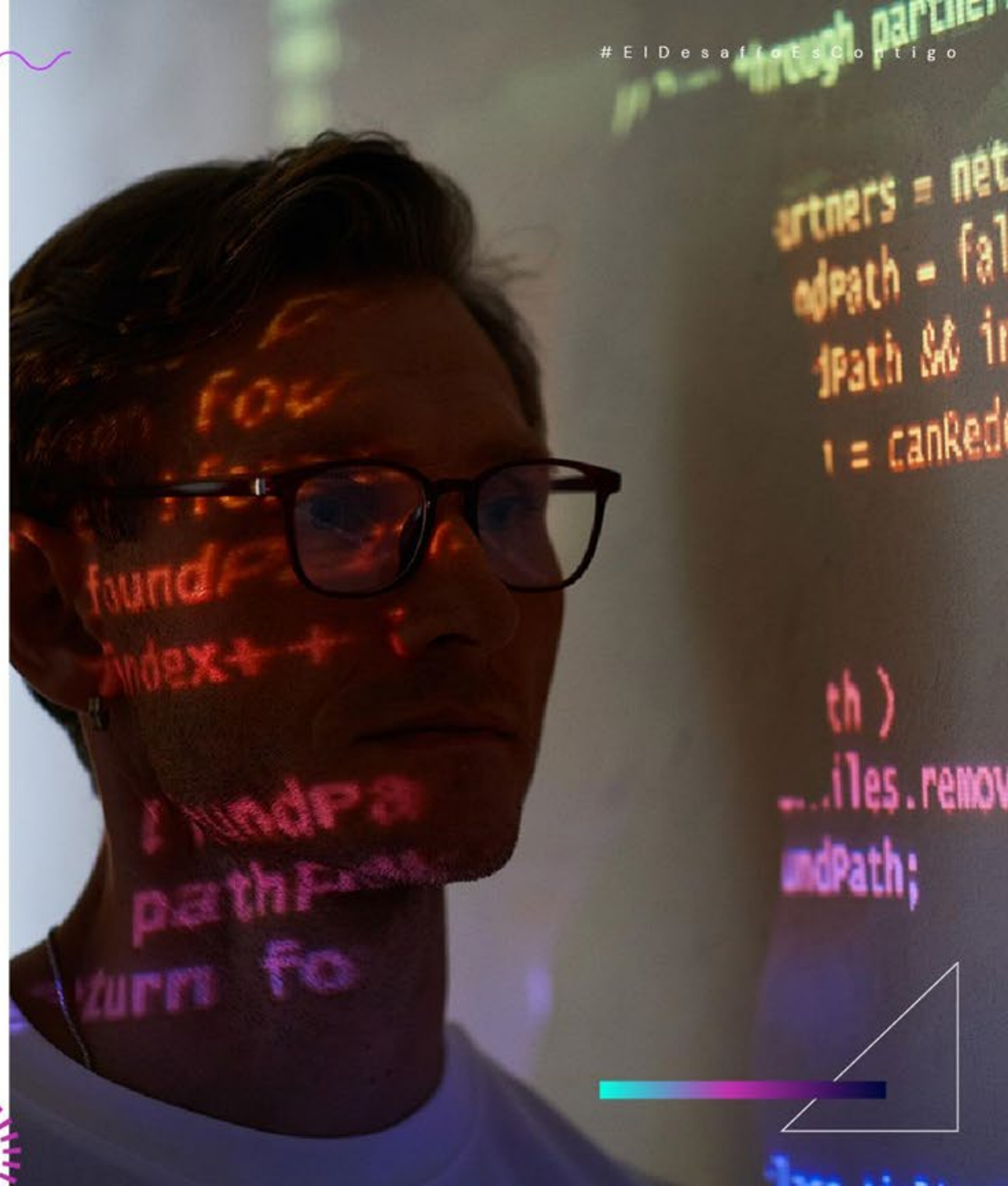
```elixir
def init(param) do
  IO.puts "Inicio GenServer GCalc"
  IO.inspect param
  :ignore
end
```



Control de errores

```elixir
def init(param) do
  IO.puts "Inicio GenServer GCalc"
  IO.inspect param
  Process.sleep(6000)
  {:ok, %{conteo: 0}}
end
```

```
iex(10)> GenServer.start_link
start_link/2    start_link/3
iex(10)> GenServer.start_link(GCalc, [], [timeout: 2500])
Inicio GenServer GCalc
[]
{:error, :timeout}
iex(11)>
```



Conclusiones

- OTP y GenServer son dos conceptos estrechamente relacionados en Elixir. OTP (Open Telecom Platform) es un conjunto de bibliotecas y herramientas para crear aplicaciones escalables y tolerantes a fallos en Erlang y Elixir.
- Mediante el uso de GenServer, los desarrolladores pueden crear procesos con estado que pueden responder a mensajes de otros procesos.

[Mishell Yagual Mendoza]
[mishell.yagual@sofka.com.co]
Technical Coach
Sofka U

Preguntas & Respuestas

Elixir

Programación más funcional que nunca

Semana 3 – MC #6

SofkaU

#ElDesafíoEsContigo

Temas

01 Supervisor

02 Application

03 Behaviors

04 Preguntas y respuestas

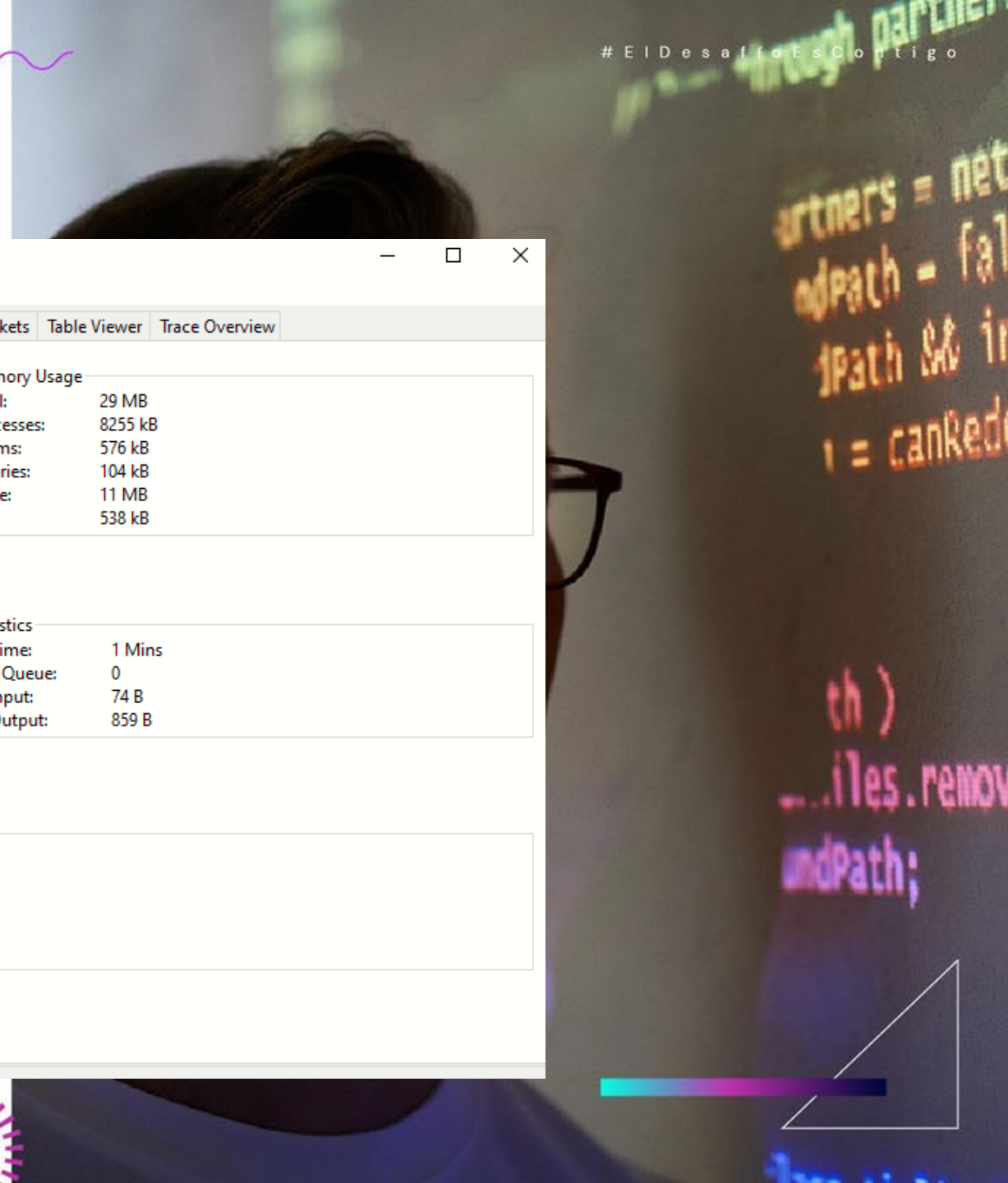
Nombrar un pid

```
iex(10)> Process.register(pid, :calculator)
true
iex(11)> send(:calculator, {self(), :x, 2, 3})
{#PID<0.106.0>, :x, 2, 3}
iex(12)> flush()
6
:ok
```

```
def add(x, y) do
  GenServer.call(GCalc.Calculator, {:add, x, y})
end

def start_link() do
  GenServer.start_link(GCalc, nil, name: GCalc.Calculator)
end
```

```
Interactive Elixir (1.11.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c("gcalc.exs")
warning: variable "reason" is unused (if the variable is not meant to be used, prefix it with an underscore)
  gcalc.exs:18: GCalc.terminate/2

warning: variable "state" is unused (if the variable is not meant to be used, prefix it with an underscore)
  gcalc.exs:18: GCalc.terminate/2

[GCalc]
iex(2)> GCalc.start_link()
Inicio GenServer GCalc
nil
{:ok, #PID<0.115.0>}
iex(3)> GCalc.add(2, 3)
handle_call(:add, 2, 3)
5
iex(4)>
```

:observer.start



nonode@nohost

File View Nodes Log Help

System | Load Charts | Memory Allocators | Applications | Processes | Ports | Sockets | Table Viewer | Trace Overview

System and Architecture

System Version:	25
ERTS Version:	13.0.4
Compiled for:	win32
Emulator Wordsize:	8
Process Wordsize:	8
SMP Support:	true
Thread Support:	true
Async thread pool size:	1

Memory Usage

Total:	29 MB
Processes:	8255 kB
Atoms:	576 kB
Binaries:	104 kB
Code:	11 MB
ETS:	538 kB

CPU's and Threads

Logical CPU's:	12
Online Logical CPU's:	12
Available Logical CPU's:	12
Schedulers:	12
Online schedulers:	12
Available schedulers:	12

Statistics

Up time:	1 Mins
Run Queue:	0
IO Input:	74 B
IO Output:	859 B

System statistics / limit

Atoms:	19678 / 1048576 (2 % used)
Processes:	71 / 262144 (0 % used)
Ports:	2 / 8192 (0 % used)
ETS:	24 / 8192 (0 % used)
Distribution buffer busy limit:	1048576

SOFKA U

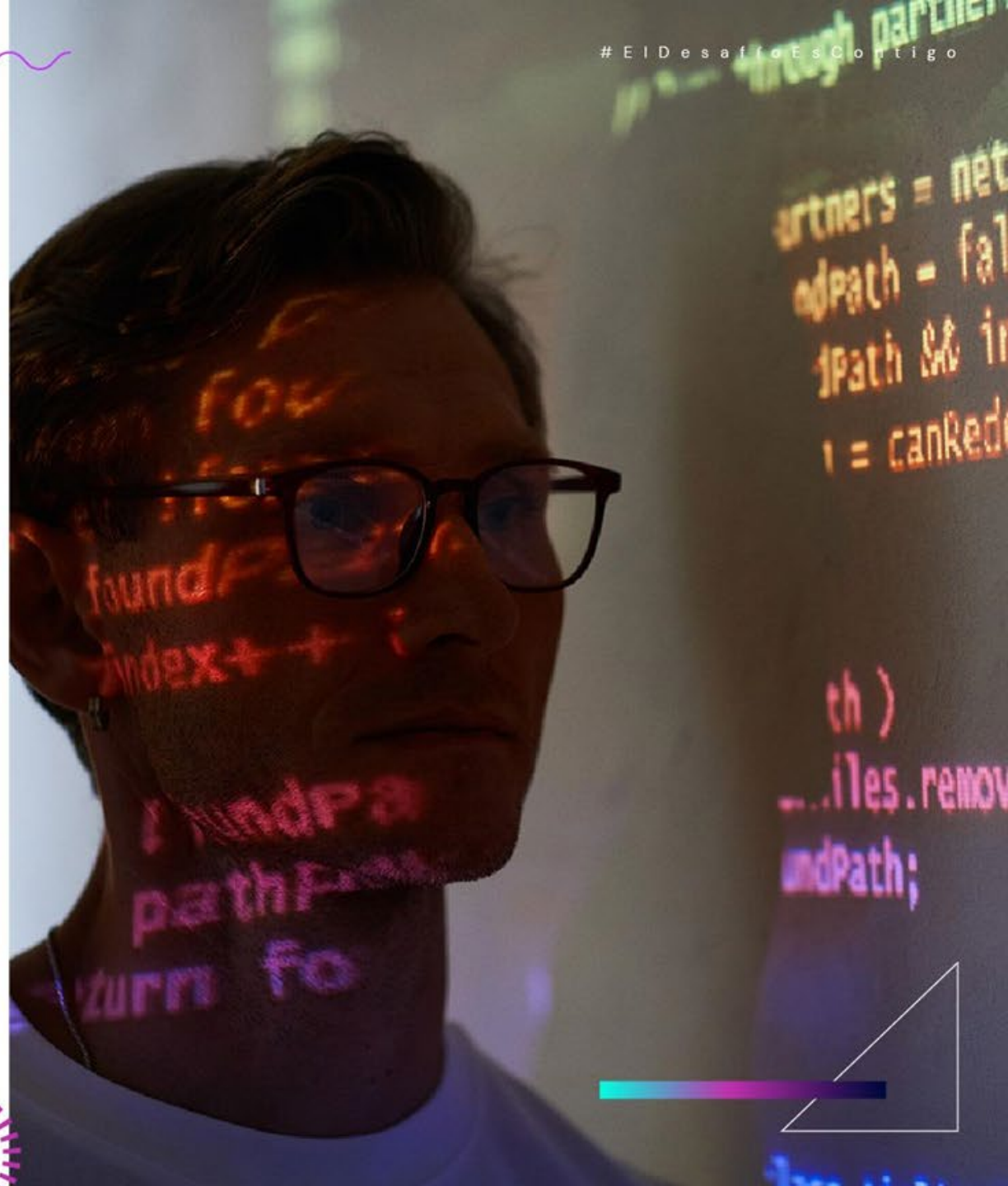
:observer.start

System | Load Charts | Memory Allocators | Applications | Processes | Ports | Sockets | Table Viewer | Trace Overview

Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.139.0>	wxe_server:init/1	756609	48816	0	gen_server:loop/7
<0.146.0>	erlang:apply/2	129707	42224	0	observer_pro_wx:table_holder/1
<0.141.0>	observer_sys_wx:init/1	36137	372328	0	wx_object:loop/6
<0.152.0>	observer_trace_wx:init/1	8432	113392	0	wx_object:loop/6
<0.149.0>	observer_sock_wx:init/1	6640	15104	0	wx_object:loop/6
<0.153.0>	erlang:apply/2	4022	2808	0	io:execute_request/3
<0.145.0>	observer_pro_wx:init/1	1509	15496	0	wx_object:loop/6
<0.150.0>	observer_tv_wx:init/1	1402	15024	0	wx_object:loop/6
<0.138.0>	observer	998	110932	0	wx_object:loop/6
<0.148.0>	observer_port_wx:init/1	979	12160	0	wx_object:loop/6
<0.112.0>	erlang:apply/2	706	2672	0	timer:sleep/1
<0.105.0>	Elixir.IEx.Evaluator:init/4	284	42312	0	Elixir.IEx.Evaluator:loop/1
<0.63.0>	user	215	8824	0	user:get_line/5
<0.115.0>	timer_server	189	7632	0	gen_server:loop/7
<0.93.0>	Elixir.Logger	42	7088	0	gen_event:fetch_msg/6
<0.46.0>	application_master:init/4	34	5920	0	application_master:main_loop/2
<0.144.0>	observer_app_wx:init/1	18	11992	0	wx_object:loop/6
<0.151.0>	erlang:apply/2	15	2672	0	observer_tv_wx:table_holder/1
<0.64.0>	erlang:apply/2	14	26592	0	Elixir.IEx.Server:wait_input/4
<0.62.0>	supervisor_bridge:user_sup/1	2	8832	0	gen_server:loop/7
<0.65.0>	kernel_config:init/1	2	2760	0	gen_server:loop/7
<0.80.0>	application_master:init/4	2	8832	0	application_master:main_loop/2
<0.86.0>	application_master:init/4	2	2800	0	application_master:main_loop/2
<0.89.0>	application_master:init/4	2	2800	0	application_master:main_loop/2
<0.94.0>	Elixir.Logger.Watcher:init/1	2	2800	0	gen_server:loop/7
<0.99.0>	Elixir.Logger.Watcher:init/1	2	2800	0	gen_server:loop/7
<0.142.0>	observer_perf_wx:init/1	2	15104	0	wx_object:loop/6
<0.143.0>	observer_alloc_wx:init/1	2	15168	0	wx_object:loop/6
<0.0.0>	init	1	42296	0	init:loop/1
<0.1.0>	erts_code_purger	1	800	0	erlang:hibernate/3
<0.2.0>	erts_literal_area_collector:start/0	1	800	0	erlang:hibernate/3
<0.3.0>	erts_dirty_process_signal_handler:...	1	2632	0	erts_dirty_process_signal_handler:msg_loop/0
<0.4.0>	erts_dirty_process_signal_handler:...	1	2632	0	erts_dirty_process_signal_handler:msg_loop/0
<0.5.0>	erts_dirty_process_signal_handler:...	1	2632	0	erts_dirty_process_signal_handler:msg_loop/0
<0.6.0>	prim_file:start/0	1	2632	0	prim_file:helper_loop/0
<0.7.0>	socket_registry	1	2632	0	socket_registry:loop/1
<0.10.0>	erl_prim_loader	1	426440	0	erl_prim_loader:loop/3
<0.42.0>	logger	1	16768	0	gen_server:loop/7
<0.44.0>	application_controller	1	689768	0	gen_server:loop/7
<0.47.0>	application_master:start_it/4	1	5720	0	application_master:loop_it/4
<0.49.0>	kernel_sup	1	19120	0	gen_server:loop/7
<0.50.0>	code_server	1	176216	0	code_server:loop/1
<0.51.0>	inet_db	1	2840	0	gen_server:loop/7
<0.52.0>	rex	1	2824	0	gen_server:loop/7
<0.53.0>	erlang:apply/2	1	2836	0	rpc:nodes_observer_loop/1
<0.54.0>	global_name_server	1	4232	0	gen_server:loop/7
<0.55.0>	erlang:apply/2	1	2672	0	global:loop_the_locker/1
<0.56.0>	erlang:apply/2	1	2672	0	global:loop_the_registrar/0

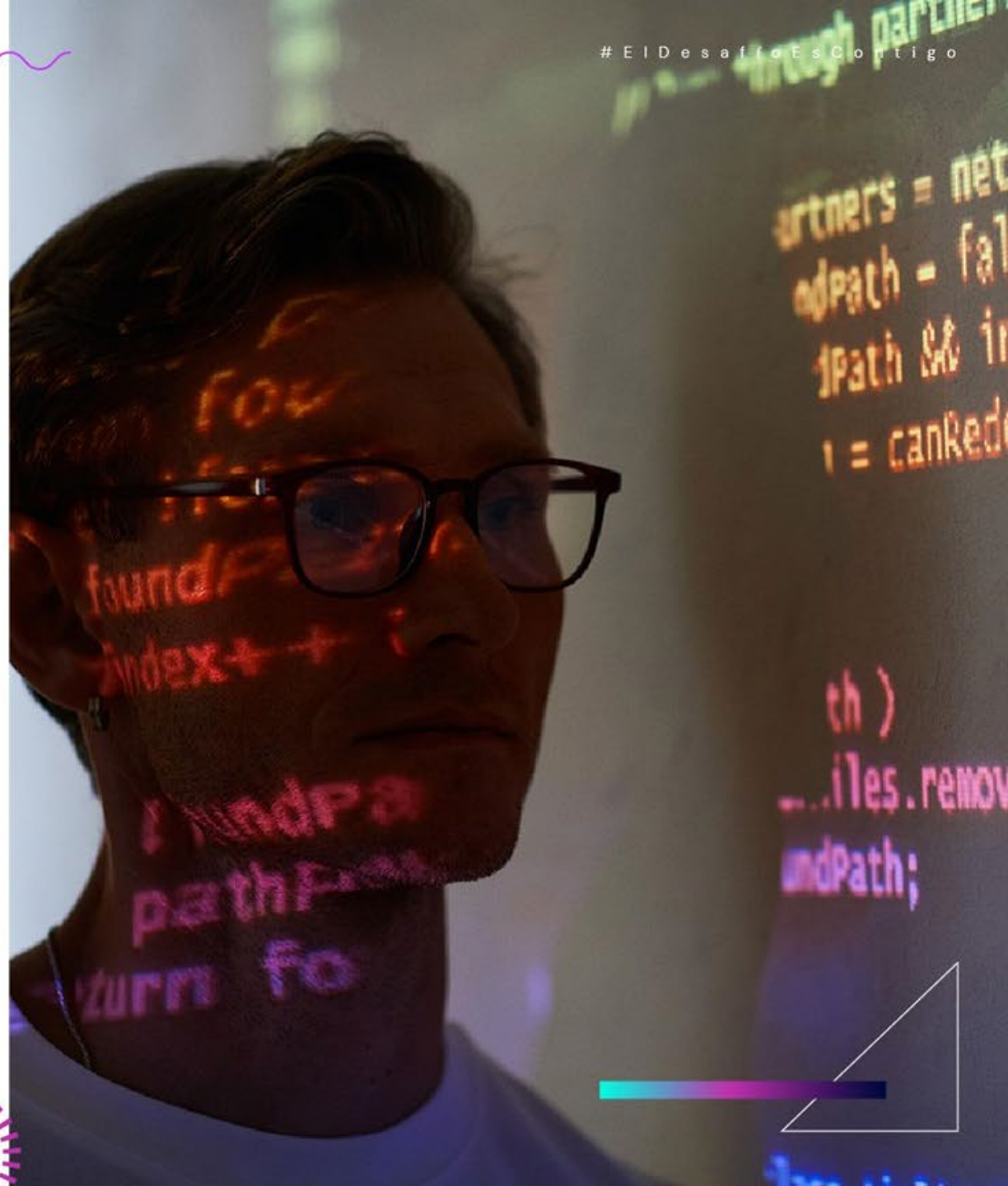
Supervisor

- Proceso responsable de iniciar, detener y monitorizar otros procesos en una aplicación Elixir.
- Los supervisores se definen mediante el comportamiento Supervisor, que es un conjunto de llamadas de retorno que definen cómo debe gestionar el supervisor sus procesos hijos.



Estrategias de reinicio

- :one_for_one: Si un proceso cae, es el único proceso que sufre un reinicio
- :one_for_all: Si un proceso cae, todos serán reiniciado (parecido a un efecto dominó)
- :rest_for_one: Si un proceso cae, los procesos que fueron iniciados después del mismo también serán reiniciados.



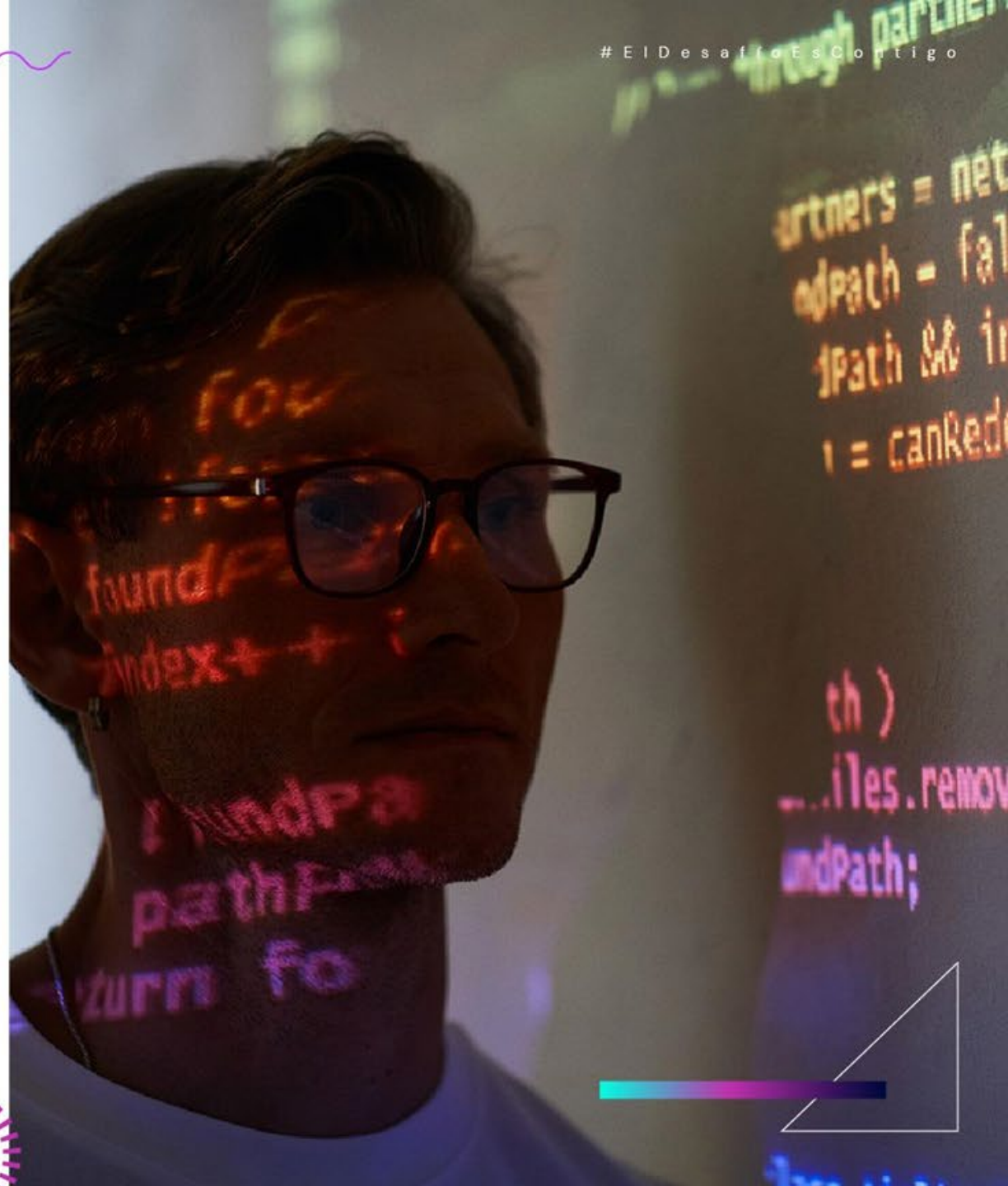
Estrategias de reinicio

```elixir
defmodule S3.Genserver.Supervisor2 do
  alias S3.Genserver.Genserver2, as: GCalculator
  use Supervisor

  def start_link([]) do
    Supervisor.start_link(__MODULE__, [], name: __MODULE__)
  end

  def init([]) do
    children = [
      {GCalculator, [] }
    ]

    # Restart the process if it crashes due to a division by zero error
    # with a maximum of 3 restarts in 10 seconds.

    Supervisor.init(children, strategy: :one_for_one, max_restarts: 3, max_seconds: 10)
  end
end
```

Aplicación

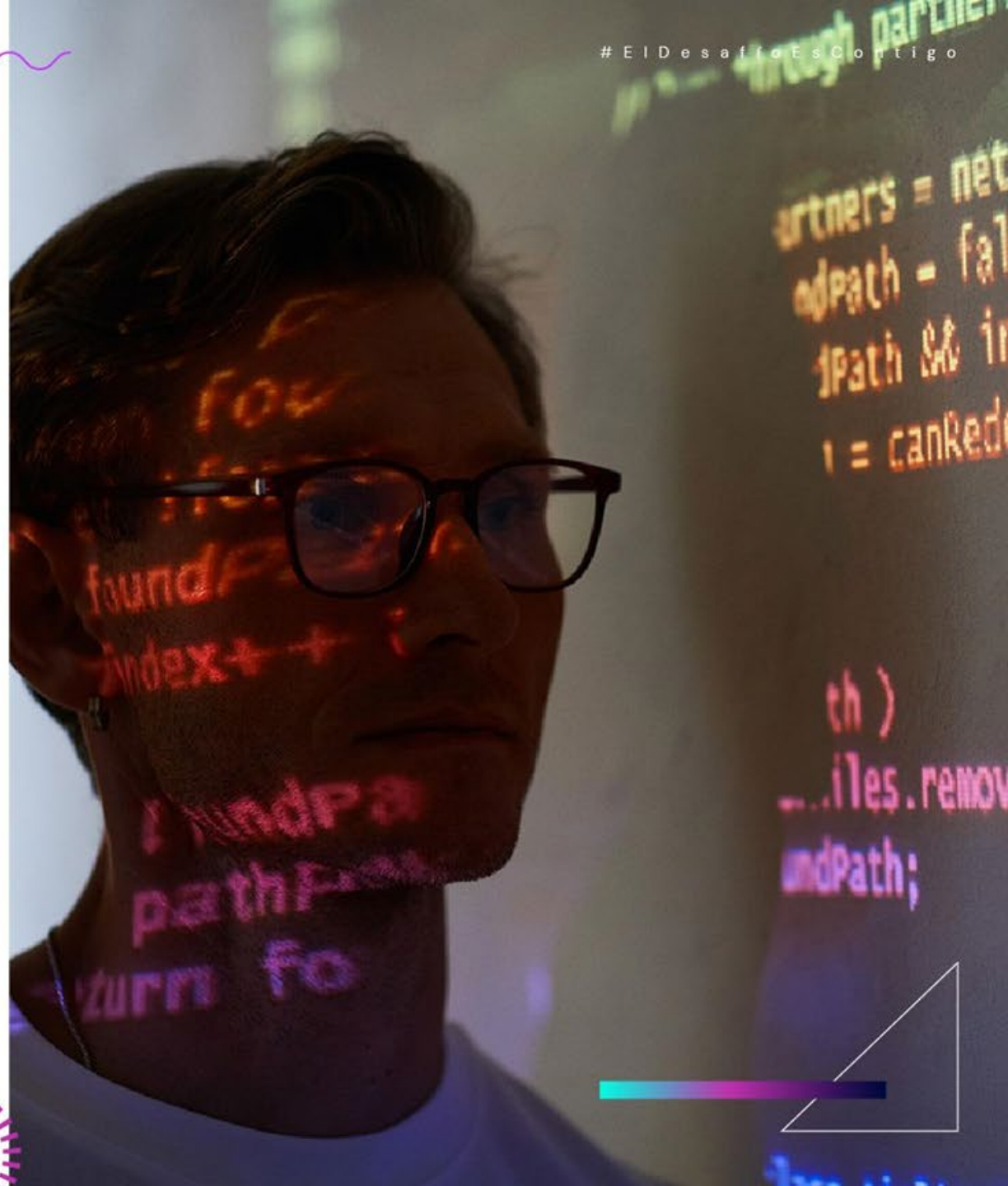
- Componente de nivel superior de un sistema Elixir que contiene uno o más supervisores y otros procesos.

- Las aplicaciones pueden iniciarse y detenerse utilizando el módulo Aplicación, que proporciona funciones para gestionar el ciclo de vida de la aplicación.



Comportamientos

- Conjunto de callbacks que definen un comportamiento específico o una interfaz para un módulo en Elixir.
- Los módulos que implementan el comportamiento GenServer pueden utilizarse indistintamente en aplicaciones Elixir, lo que facilita la creación y el mantenimiento de sistemas complejos.

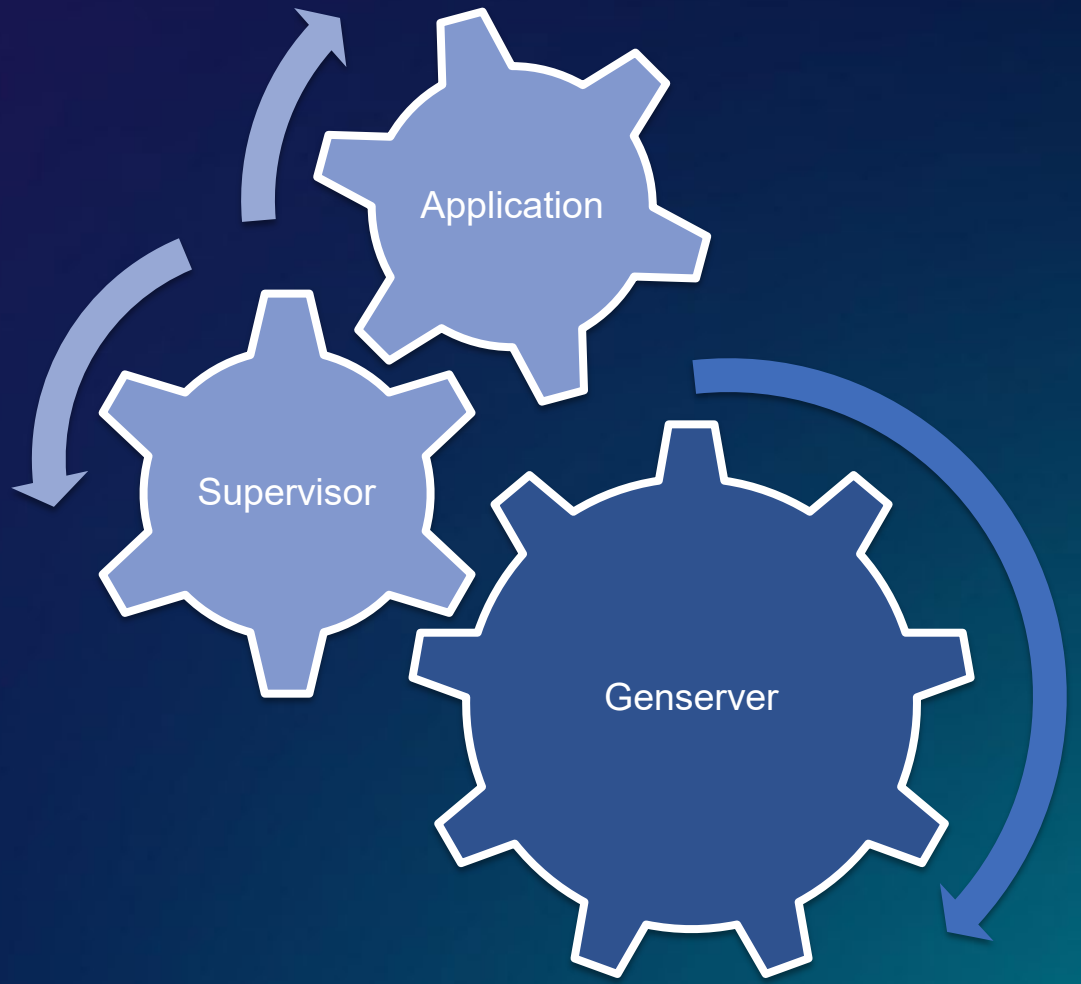


¿Cómo se relacionan estos conceptos entre sí?

GenServer es un comportamiento que suele utilizarse para implementar la lógica de negocio de una aplicación

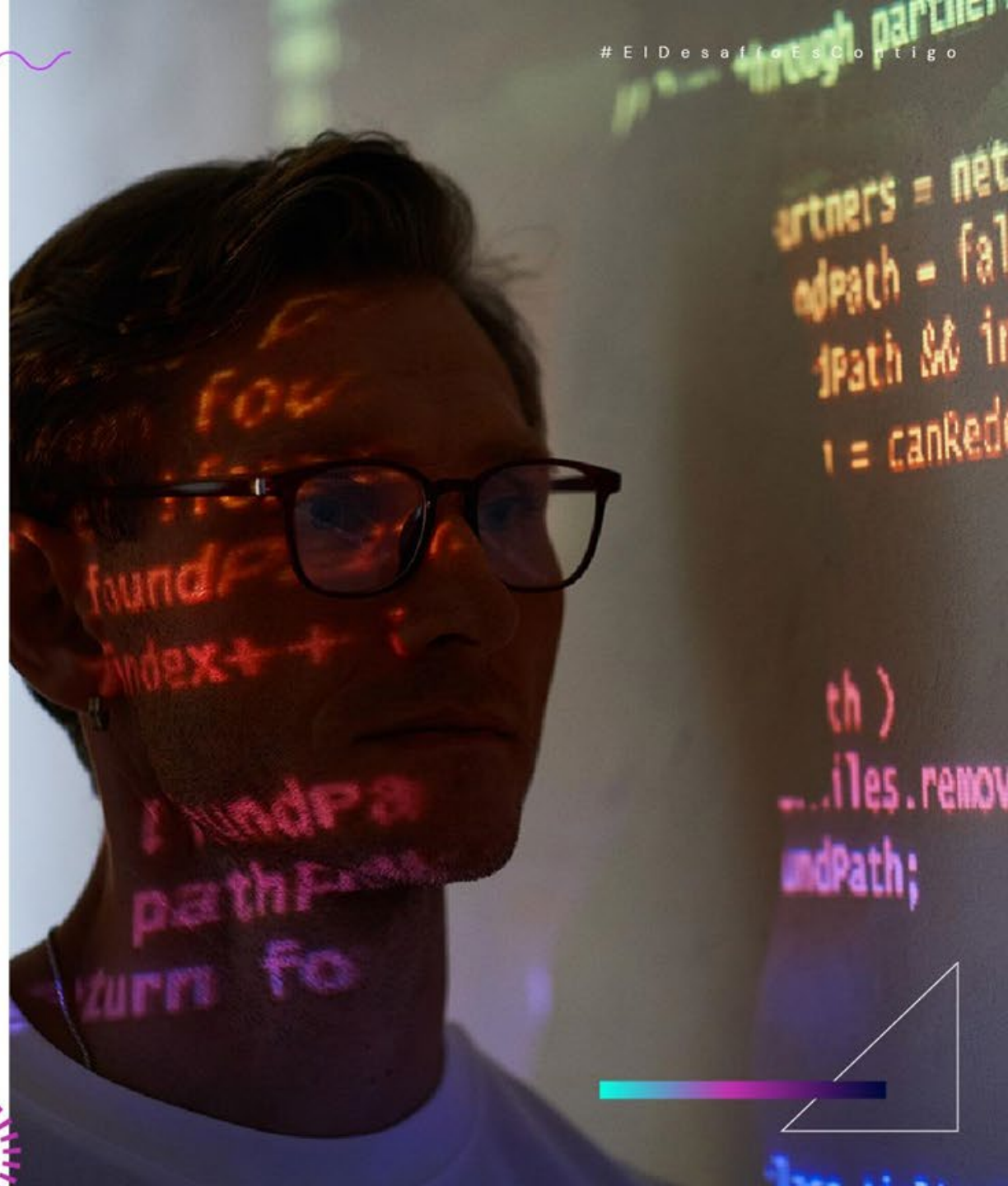
Supervisor es un proceso responsable de gestionar el ciclo de vida de uno o varios procesos GenServer.

Una Aplicación es un componente de nivel superior que contiene uno o más supervisores y otros procesos.



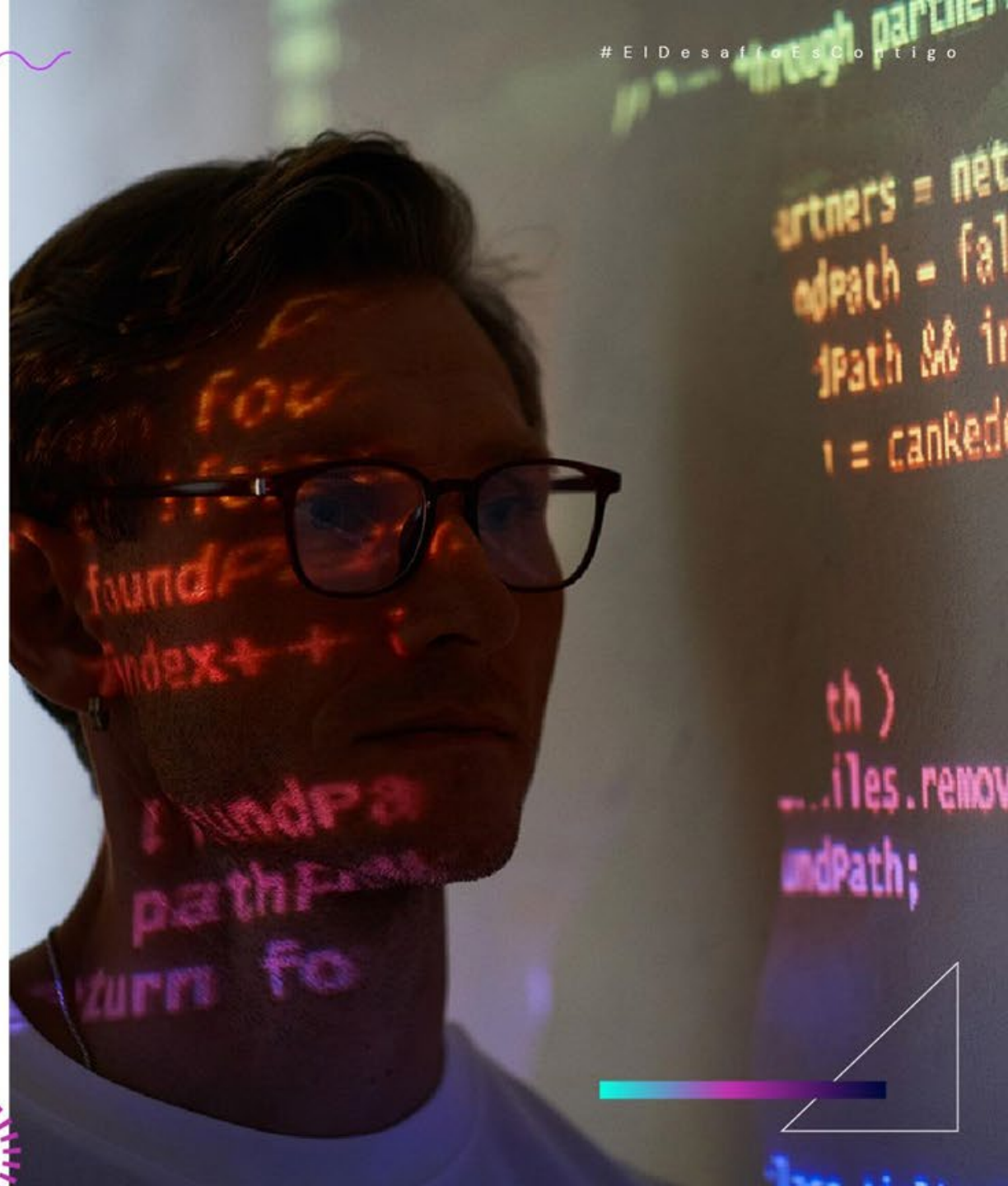
Ejemplo

```
defmodule Counter do
  use GenServer

  def start_link(init_count) do
    GenServer.start_link(__MODULE__, init_count)
  end

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

  def handle_call(:increment, _from, count) do
    {:reply, count + 1, count + 1}
  end

  def handle_call(:decrement, _from, count) do
    {:reply, count - 1, count - 1}
  end
end
```



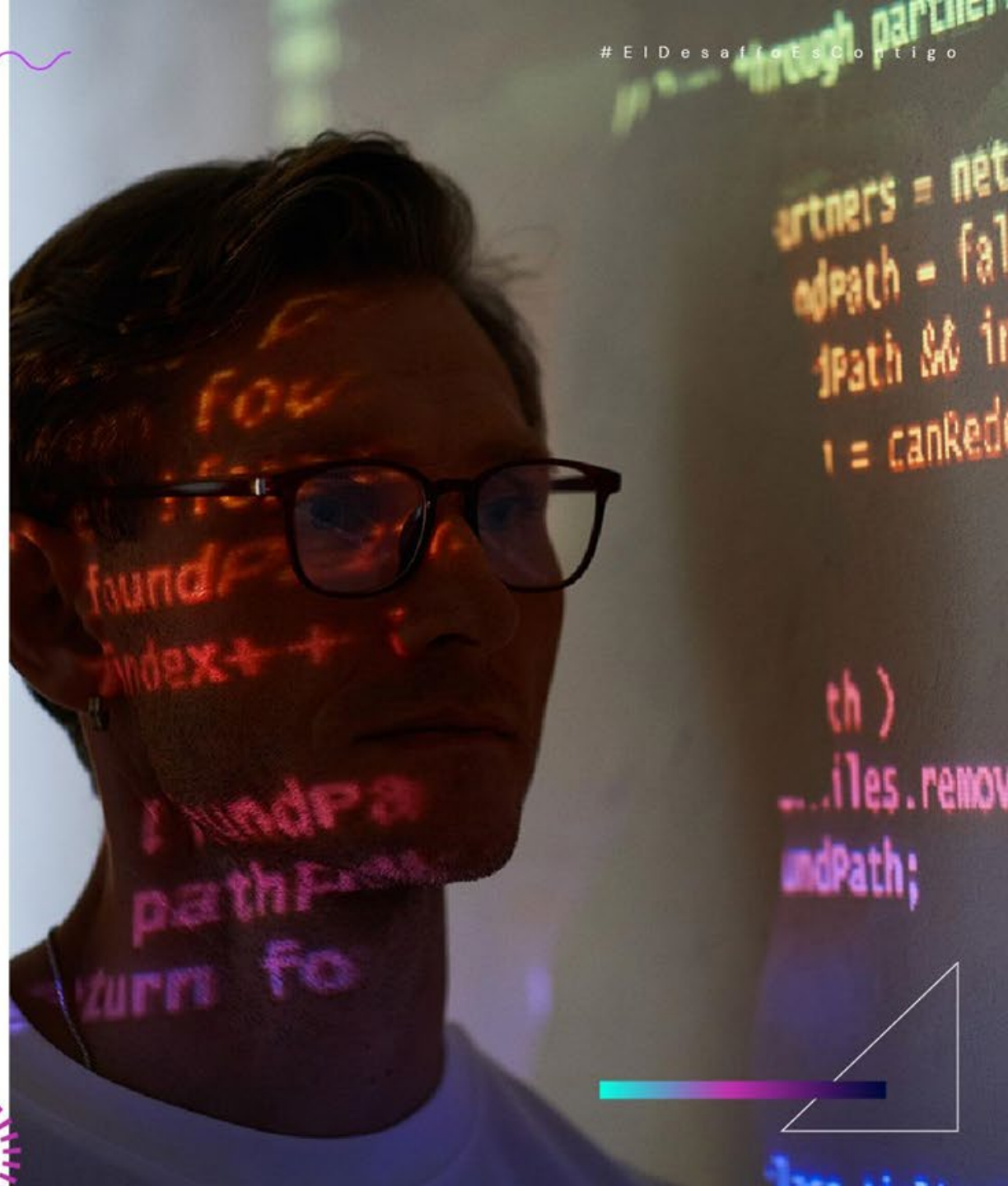
Ejemplo

```
defmodule Counter.Supervisor do
  use Supervisor

  def start_link(init_count) do
    Supervisor.start_link(__MODULE__, init_count)
  end

  def init(init_count) do
    children = [
      worker(Counter, [init_count])
    ]
    supervise(children, strategy: :one_for_one)
  end
end
```



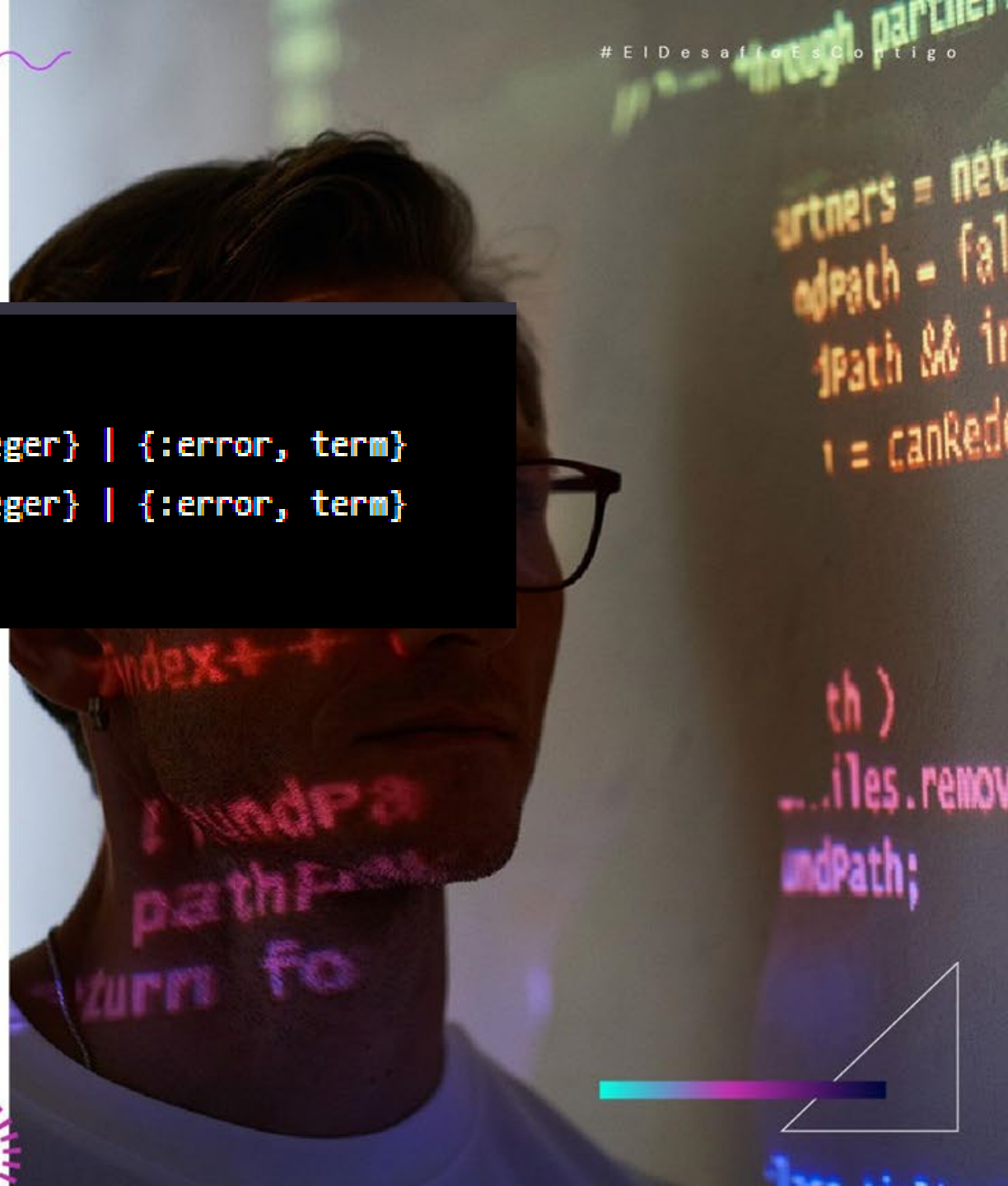
Ejemplo

```
defmodule CounterApp do
  use Application

  def start(_type, _args) do
    init_count = 0
    Counter.Supervisor.start_link(init_count)
  end
end
```



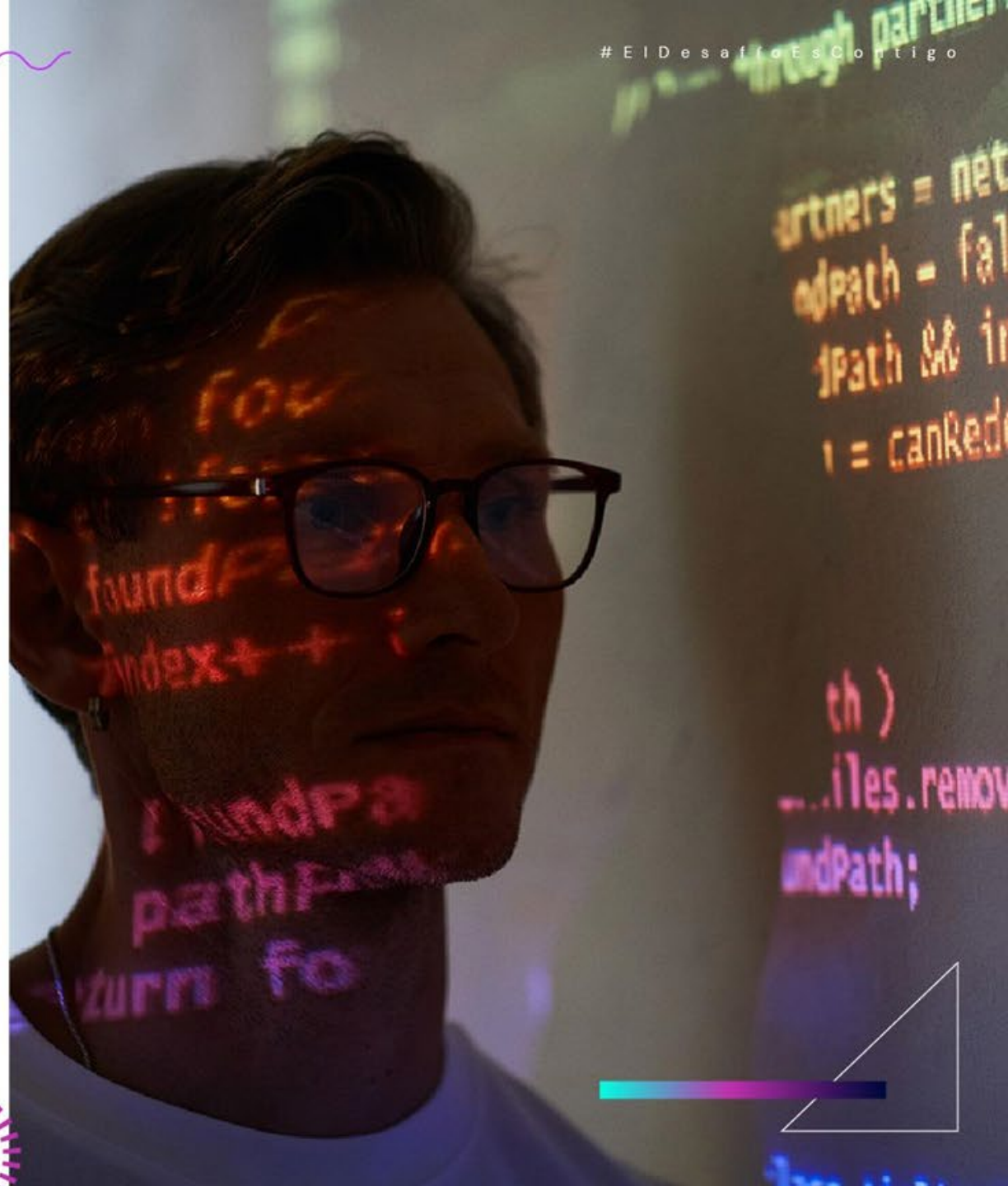
Ejemplo

```
defmodule Counter.Behavior do
  @callback increment(integer) :: {:ok, integer} | {:error, term}
  @callback decrement(integer) :: {:ok, integer} | {:error, term}
end
```



Ejemplo

```
defmodule MyModule do
  use GenServer
  use Counter.Behavior

  def start_link(init_count) do
    GenServer.start_link(__MODULE__, init_count)
  end

  def handle_call(:increment, _from, state) do
    {:reply, Counter.increment(state), state}
  end

  def handle_call(:decrement, _from, state) do
    {:reply, Counter.decrement(state), state}
  end
end
```



Conclusiones

- Los supervisores son procesos responsables de iniciar, detener y supervisar otros procesos, incluidos los GenServers.
- Una aplicación es un componente de nivel superior que contiene uno o más supervisores y otros procesos, incluidos los GenServers, y proporciona una función de devolución de llamada que inicia los procesos de la aplicación.
- Los comportamientos son un conjunto de retrollamadas que definen un comportamiento específico o una interfaz para un módulo en Elixir.

[Mishell Yagual Mendoza]
[mishell.yagual@sofka.com.co]
Technical Coach
Sofka U

Preguntas & Respuestas

#ElDesafíoEsContigo

Calle 12 # 30-80 Medellín

Calle 85 # 11 – 53 Int 6 Of. 301 Bogotá

+57 604 266 4547

info@sofka.com.co

www.sofka.com.co

Síguenos

Sofka Technologies

Sofka_Technologies