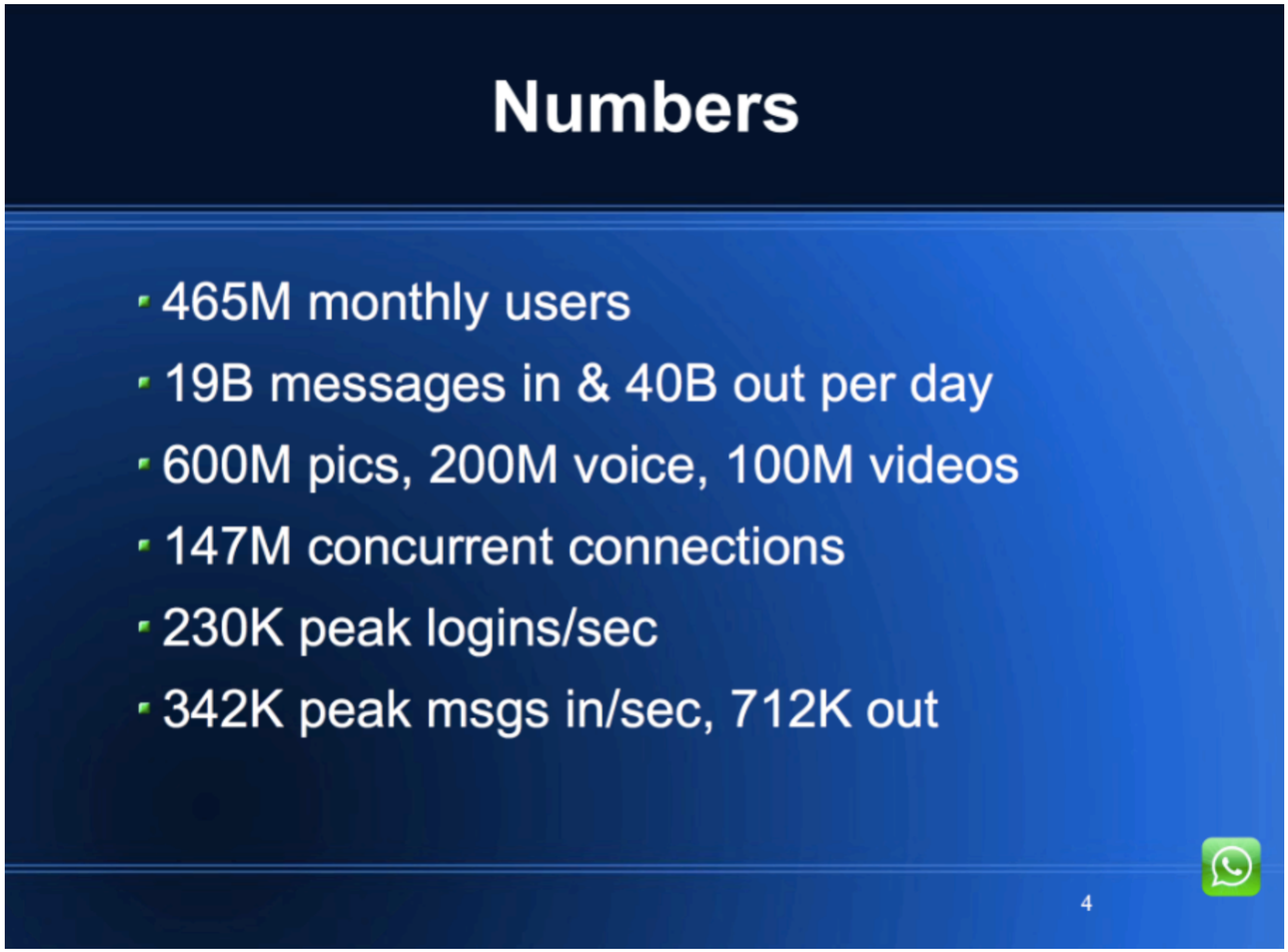
# Lo mejor de dos mundos: OTP y Elixir

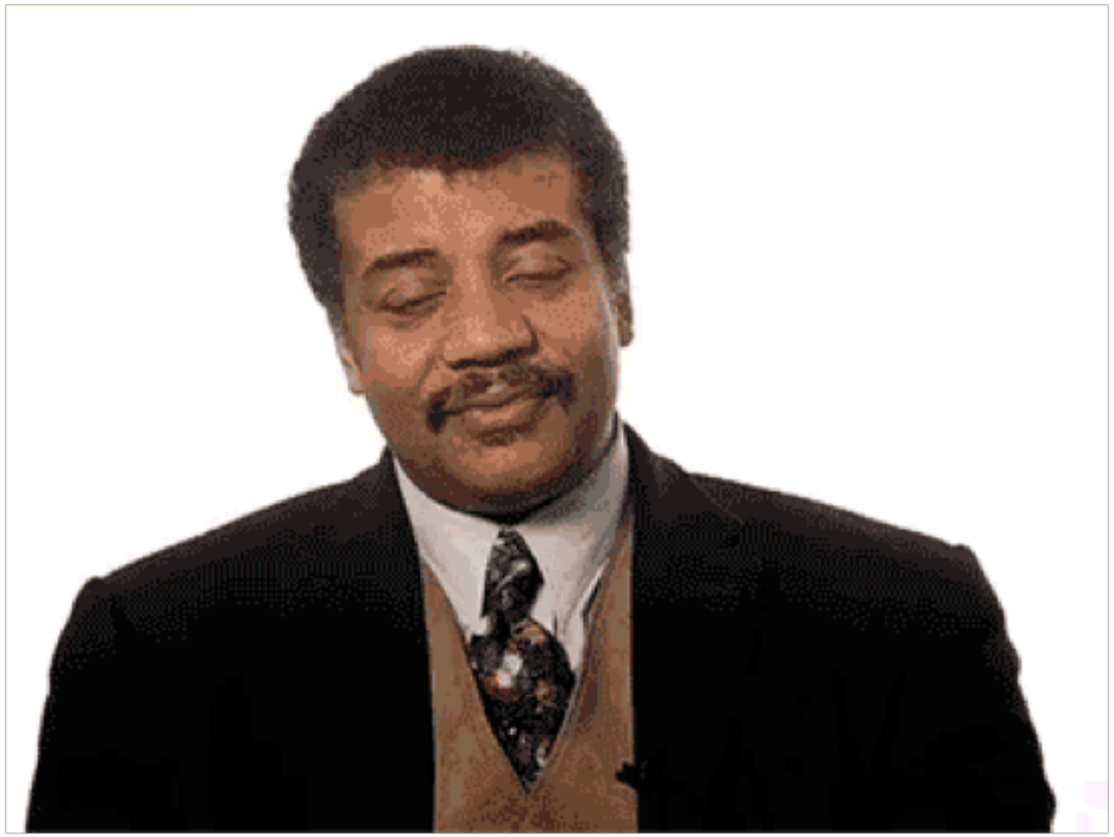
Erick Camacho @ecamacho


# Concurrencia basada en proceso

# Tolerancia a fallos

# Escalabilidad





# Desde Elixir, ¿cómo sacar provecho de Erlang?

# Si ya uso Erlang, ¿qué me puede dar Elixir?

¿OTP?


Outlaw Techno
Psychobitch!

**Paul Wilson**
@paulanthonywils

Follow

“What does OTP stand for?”
“Other Than Phoenix”

- paraphrase @jimfreeze #ElixirConf2015

Framework para construir aplicaciones estables, concurrentes y tolerantes a fallos

# Nuestro caso de estudio


elixir
TELCEL
9:24
sábado, 3 de octubre
Nubleer
Hi from Elixir

# Construimos un módulo que hace esa tarea

ApnsConnection

```
def connect(environment, {certpath, keypath}) do
#Regresa una conexión al servidor de Apple

def send(conn, device_token, message) do
#Envía una notificación push al dispositivo, usando la conexión
```

Podemos simplemente invocarlo desde un Controller en nuestra app web, pero:

¿Qué pasa si el servidor de Apple se cae?

¿Qué pasa si queremos enviar un mensaje a 10 mil dispositivos?

Vamos a extraer esa funcionalidad a una aplicación OTP.

# Aplicaciones

Conjunto de módulos + metadatos (para versionamiento, por ejemplo).
Tienen un ciclo de vida definido: **start**, **stop**.

# Con Mix puedes gener una Aplicación

```
mix new misiva
mix compile
Generated misiva app
```

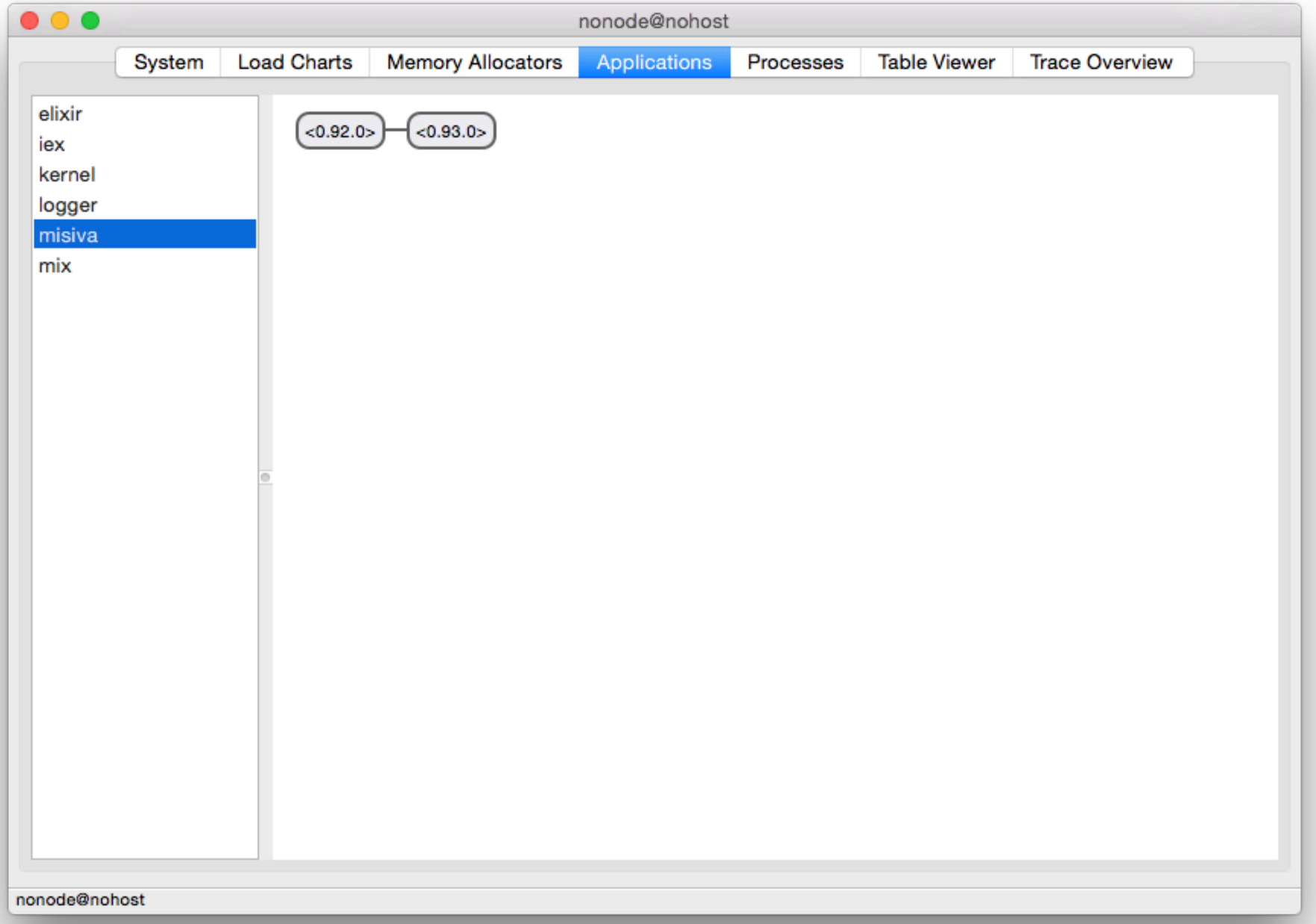
# En una app OTP conviene crear un Application Callback

# en lib/misiva.ex

```
defmodule Misiva do
 use Application
 require Logger

 def start(_type, _args) do
  Logger.info "starting app"
  {:ok, self}
 end

 def stop(_state) do
  Logger.info "stopping app"
  :ok
 end

end
```

En mix.exs

```
def application do
  [mod: {Misiva, []},
  applications: [:logger]]
end
```


nonode@nohost
System
Load Charts
Memory Allocators
Applications
Processes
Table Viewer
Trace Overview
elixir
iex
kernel
logger
misiva
mix
<0.92.0>
<0.93.0>
nonode@nohost

# ¿Cómo creamos un proceso para enviar nuestros mensajes?

# The old spawn_link

```
spawn_link fn -> 1 + 2 end
```

# O usar un Task

```
Task.start fn -> raise "oops" end
```

# Además necesitas un loop para conservar el estado

```
defp loop(map) do
  receive do
    {:get, key, caller} ->
      send caller, Map.get(map, key)
      loop(map)
    {:put, key, value} ->
      loop(Map.put(map, key, value))
  end
end
```

# GenServer

- OTP *Behaviour*
- Abstrae la funcionalidad de un **servidor**
- No necesitas lidiar con mensajes
- No necesitas lidiar con un loop
- Mantiene estado

# ¿Cómo? Implementas unos callbacks

```
use GenServer

def init(_) do

def handle_call(_) do

def handle_cast(_) do

def terminate(_) do
```

# init/1

Llamado cuando se invoca **start_link**

Regresa {:ok, state}

State es el estado del proceso. El estado se mantendrá entre llamadas.

# handle_call/3

Recibe una petición y contesta de forma síncrona

Regresa
- {:reply, respuesta, state} La respuesta es enviada a quien llamó la función
- {:noreply, state} Quien llamó la función no recibe respuesta
- {:stop, state} Termina el gen_server y se invoca terminate/2

# handle_cast/2

Recibe una petición y la procesa de forma asíncrona

Regresa
- {:noreply, state} Quien llamó la función no recibe respuesta
- {:stop, state} Termina el gen_server y se invoca terminate/2

# terminate/2

Llamado cuando el gen_server va a terminarse. Sirve para liberar recursos.

# init

```
defmodule Misiva.Apns do
  use GenServer

  def start_link(args, opts \\ []) do
    GenServer.start_link(__MODULE__, args, opts)
  end

  def init([{env, certpath, keypath}]) do
    case Misiva.ApnsConnection.connect(env, {certpath, keypath}) do
      {:ok, conn} ->
        { :ok, conn}
      {:error, reason} ->
        {:error, reason}
    end
  end
```

# handle_call

```
def send(pid, token, message) do
  GenServer.call(pid, {:send, token, message})
end

def handle_call({:send, token, message}, _from, conn) do
  case Misiva.ApnsConnection.send(conn, token, message) do
    {:ok, _} ->
      {:reply, :ok, conn}
    {:error, _} ->
      {:reply, :error, conn}
  end
end
```

# terminate

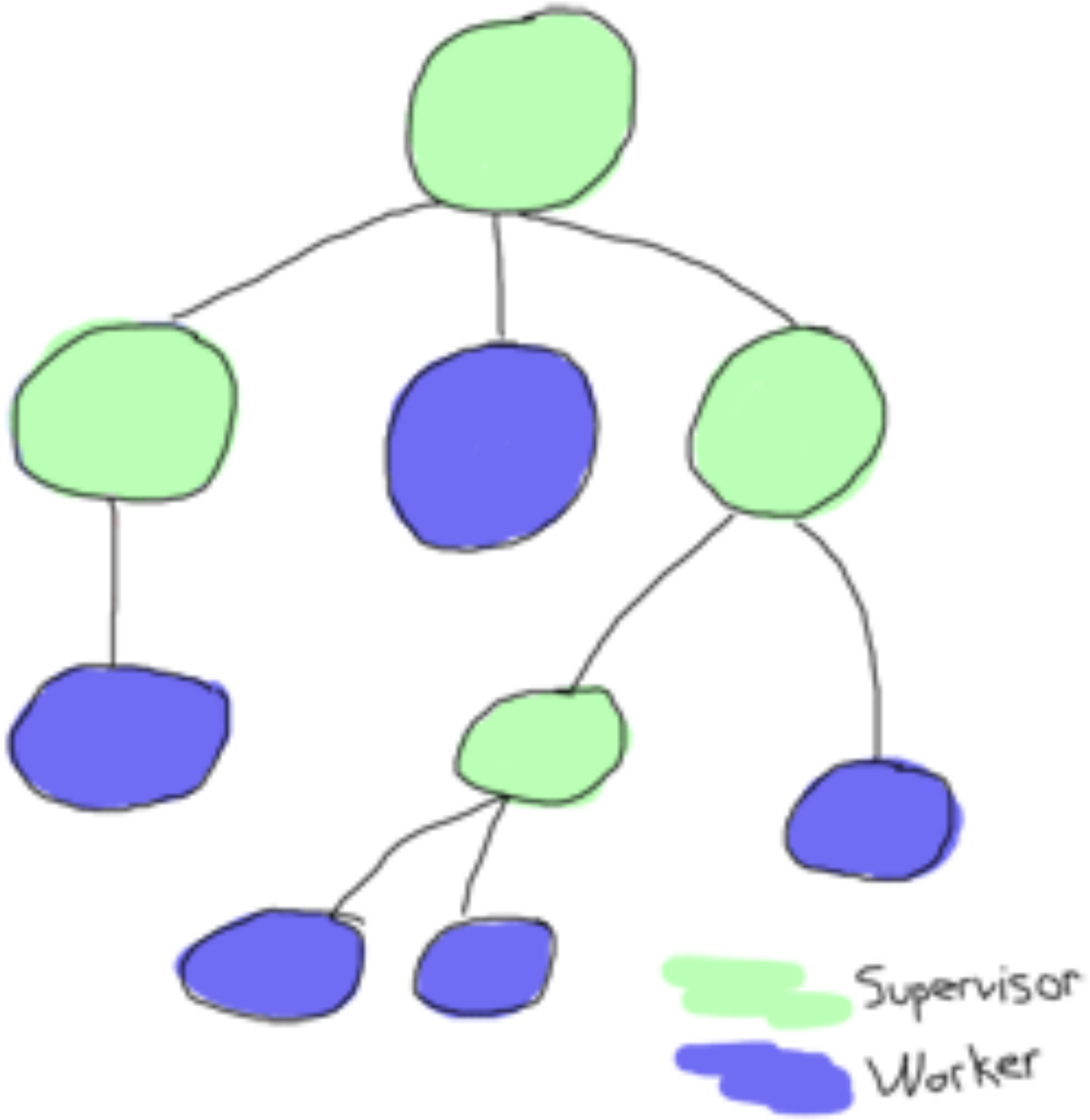
```
def terminate(_reason, conn) do
    Misiva.ApnsConnection.close(conn)
end
```

# ¿Qué pasa si se cae el gen_server?

# Una opción es monitorear el proceso

```
Process.monitor pid
```

# Supervisors


Supervisor
Worker

En OTP **todos** los procesos (*workers*) deben ser *supervisados*
El *Supervisor* :
- Inicia los procesos
- Si se caen, puede volver a levantarlos
- Cuando la aplicación se termina, se encarga de terminar correctamente todos los procesos

# Una aplicación OTP es un árbol de supervisores y workers

# Podemos crear un módulo nuevo

```
defmodule Misiva.Supervisor do
    use Supervisor

    ...
```

# O en nuestro App callback crear un Supervisor.Spec

```
defmodule Misiva do
 use Application
 require Logger

 def start(_type, _args) do
  import Supervisor.Spec, warn: false
  args = {:production, "/etc/cert.pem", "/etc/key.pem"}
  children = [
    worker(Misiva.Apns, [args], [name: :apns1])
  ]
  opts = [strategy: :one_for_one, name: Misiva.Supervisor]
  Supervisor.start_link(children, opts)
 end

 def stop(_state) do
  Logger.info "stopping app"
  :ok
 end

end
```

# Restart Strategy

- one for one
- one for all
- rest for one
- simple one for one

# Restart

- permanent: Siempre se reinicia
- temporary: Nunca se reinicia
- transient: Se reinicia sólo cuando el proceso terminó con una razón diferente a :normal o :shutdown

# Tip

Puedes crear una app con un Supervisor listo con el parámetro *--sup* con mix

```
mix new myapp --sup
```

# Supervisión dinámica

En lugar de tener un número fijo de procesos, puedes pedirle al Supervisor que inicie uno en cualquier momento y lo agregue a su árbol de supervisión

```elixir
defmodule Misiva do
 use Application
 require Logger

 def start(_type, _args) do
  import Supervisor.Spec, warn: false
  children = [
    worker(Misiva.Apns, [], [restart: :transient])
  ]
  opts = [strategy: :simple_one_for_one, name: Misiva.Supervisor]
  Supervisor.start_link(children, opts)
 end

 def stop(_state) do
  Logger.info "stopping app"
  :ok
 end

end
```

# Ahora puedes crear workers conforme los necesites

```
{:ok, pid} = Supervisor.start_child(Misiva.Supervisor,
[[args], [name: :apns1]])

{:ok, pid2} = Supervisor.start_child(Misiva.Supervisor,
[[args], [name: :apns2]])

{:ok, pid3} = Supervisor.start_child(Misiva.Supervisor,
[[args], [name: :apns3]])
```

# Tip: las *Tasks* pueden ser supervisadas

```
children = [
  worker(Task, [fn -> IO.puts "ok" end])
]
```

# Estamos listos para integrarlo a nuestra app Phoenix

En el mix.exs de tu app web

1. Agrega la dependencia

2. Agrega la app *misiva* a la lista de aplicaciones

Mix se encargará de iniciar *misiva* cuando inicie tu app

```
defmodule Myapp.Mixfile do
  use Mix.Project

  def application do
    [mod: {Myapp, []},
     applications: [:phoenix, :phoenix_html, :cowboy, :logger,
                    :phoenix_ecto, :postgrex, :misiva]]
  end

  defp deps do
    [{:phoenix, "~> 1.0.3"},
     {:phoenix_ecto, "~> 1.1"},
     {:postgrex, ">= 0.0.0"},
     {:phoenix_html, "~> 2.1"},
     {:phoenix_live_reload, "~> 1.0", only: :dev},
     {:cowboy, "~> 1.0"},
     {:misiva, "~> 0.0.1", path: "/path/to/misiva"}
    ]
  end

end
```

# OTP Tiene más behaviours:

- gen_event
- gen_fsm
- gen_tcp

# Usa OTP

- OTP Design principles http://www.erlang.org/doc/design*principles/des*princ.html
- Elixir Introduction to Mix http://elixir-lang.org/getting-started/mix-otp/introduction-to-mix.html

# Learn You Some Erlang for Great Good!

## A Beginner's Guide

Fred Hébert

# THE LITTLE Elixir & OTP GUIDEBOOK

Benjamin Tan Wei Hao

MANNING

¡Misiva es Open Source!

http://github.com/nubleer/misiva

Gracias @ecamacho