ELIXIR

Oscar Fabián Ñáñez Núñez
Gabriela Suárez Carvajal
Sergio Alexander Gil

CONCEPTOS BÁSICOS DE ELIXIR

“Elixir es un lenguaje dinámico y funcional diseñado para construir aplicaciones escalables y mantenibles.” — elixir-lang.org

HISTORIA

Elixir surgió hace poco, en el 2012 José Valim decidió crearlo porque había notado falencias en las plataformas que utilizaba en aspectos de concurrencia y en la resistencia a fallos, decidió entonces basarse en la maquina virtual de erlang y seguir la interfaz de RoR

1. SALIDA

Para imprimir en la salida de la consola, se utiliza la siguiente expresión:

```
IO.puts "Hello world"
```

o con paréntesis:

```
IO.puts("Hello world")
```

2. BASES

2.2 Comentarios

Comentarios de una línea -> #

```
#This is a comment in Elixir
```

Comentarios de varias líneas-> no existen

2.3 Finales de línea

No es obligatorio poner ningun caracter como fin de línea.

2.4 Identificadores

Se empieza con letras minúsculas y luego se pueden poner numeros, _ o letras mayúsculas.

```
var1  variable_2  one_M0r3_variable
```

Un valor que no debe ser usado se asigna a una variable que empiece con _. Es decir esa variable que no se usará nuevamente, pero es necesario asignarla a algo. También se puede hacer con el nombre de una función.

```
_some_random_value = 42
```

2.5 Palabras reservadas

after	and	catch	do
inbits	inlist	nil	else
end not	or	false	fn	in
rescue	true	when	xor
__MODULE__	__FILE__	__DIR__
__ENV__	__CALLER__

2.5. Tipos de datos.

Al igual que otros lenguajes, tiene los tipos de datos básicos (tabla 1). Junto con un tipo de dato propio de Elixir conocido como *“atom”*.

Atom: son constantes cuyo nombre es su valor. Se crean usando el símbolo `:`.

Ejemplo:

```
:hello
```

integers	
floats	Elixir admite precisión de 64 bits para números flotantes. También se pueden definir utilizando un estilo de exponenciación. Ejemplo: 10145230000 se puede escribir como 1.014523e10.

Strings	Se insertan entre comillas simples.
	`"Hello world"`
	Y para definir múltiples strings:
	`"""`
	`Hello`
	`World!`
	`"""`

Booleanos	Elixir admite **true** y **false** como booleanos. Ambos valores están de hecho unidos a átomos `:true` y `:false` respectivamente.

Tabla 1

Octal	Para definir un número en la base octal se usa el prefijo '0o'. Ejemplo: 0o52 en octal equivale a 42 en decimal.
Hexadecimal	Para definir un número en base decimal se usa el prefijo '0x'. Ejemplo: 0xF1 en hex es equivalente a 241 en decimal.
Binario	Para definir un número en base binaria se usa el prefijo '0b'. Ejemplo: 0b1101 en binario es equivalente a 13 en decimal.
Binarios	Los binarios se usan principalmente para manejar datos relacionados con bits y bytes. `<< 65, 68, 75>>`
Listas	Elixir usa corchetes para especificar una lista de valores. Los valores pueden ser de cualquier tipo. `[1, "Hello", :an_atom, true]`
Tuplas	Elixir usa corchetes para definir tuplas. Al igual que las listas, las tuplas pueden contener cualquier valor. `{ 1, "Hello", :an_atom, true`

Tabla 1

Diferencia entre tuplas y listas.

	Organizado como	Insertar	Eliminar	Acceder
Lista	linked list	very fast	very fast	
Tupla	bloques de memoria continua			very fast

2.6. Declaración de variables.

Una variable debe declararse y asignarse un valor al mismo tiempo.

```
life = 42
```

Si queremos reasignar a esta variable un nuevo valor, podemos hacer esto usando la misma sintaxis.

```
life = "Hello world"
```

2.7. Operadores.

Operadores aritméticos Supongamos que la variable A es 10 y la variable B es 20.		
Operador	**Descripción**	**Ejemplo**
+		A + B da 30
-		A + B da -10
*		A * B da 200
/	Divide el primer número entre el segundo. Esto arroja el resultado en flotante.	A / B da 0.5
div	Esta función se usa para obtener el **cociente** en la división.	div(10, 20) da 0
rem	Esta función se usa para obtener el **residuo** en la división.	rem(A, B) da 10

Operadores de comparación Supongamos que la variable A es 10 y la variable B es 20		
Operador	**Descripción**	**Ejemplo**
==		A == B da false
!=		A != B da true
===	Comprueba si el tipo de valor a la izquierda es igual al tipo de valor a la derecha, en caso afirmativo, verifique lo mismo para el valor.	A === B da false
!==	Lo mismo que arriba, pero verifica la desigualdad en lugar de la igualdad.	A !== B da true
>		A > B da false
<		A < B da true
>=		A >= B da false
<=		A <= B da true

Operadores lógicos Suponer que la variable A es verdadera y la variable B es20		
Operador	**Descripción**	**Ejemplo**
and	Comprueba si los dos valores proporcionados son verdaderos, si es así, devuelve el valor de la segunda variable.	A and B da 20
or	Comprueba si el valor proporcionado es verdadero. Devuelve el valor que sea verdadero. Else devuelve falso.	A or B da true
not	Operador unario que invierte el valor de la entrada dada.	not A dará falso
&&	No es un and estricto. Funciona igual pero no espera que el primer argumento sea un booleano.	B && A da 20
||	No es un or estricto. Funciona igual pero no espera que el primer argumento sea booleano.	B || A da true
!	No es un no estricto. Funciona igual que pero no espera que el argumento sea booleano.	!A da false

3. La coincidencia de patrones.

Un match tiene 2 partes principales, un lado izquierdo y un lado derecho. El lado derecho es una estructura de datos de cualquier tipo. El lado izquierdo intenta hacer coincidir la estructura de datos en el lado derecho y unir las variables de la izquierda a la subestructura respectiva de la derecha.

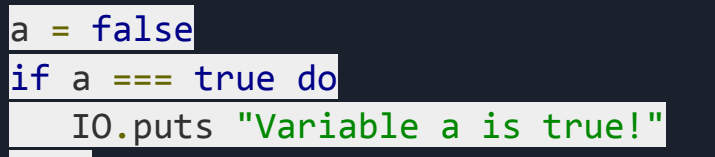
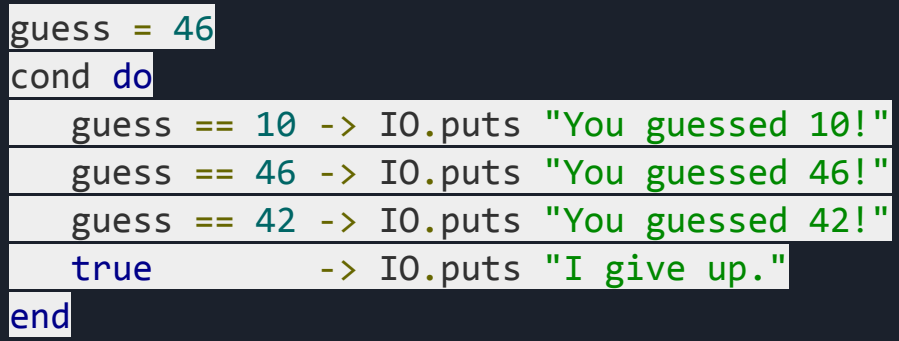
```
[var_1, _unused_var, var_2] = [{"First variable"}, 25, "Second variable" ]
```

```
[_, [_, {a}]] = ["Random string", [:an_atom, {24}]]
```

```
a = 25
b = 25
^a = b
```

Si tenemos un conjunto que no coincidente de lado izquierdo y derecho, el operador de coincidencia genera un error. Por ejemplo, si tratamos de hacer coincidir una tupla con una lista o una lista de tamaño 2 con una lista de tamaño 3, se mostrará un error.

3. Toma de decisiones.

if else statement	Una instrucción if puede ser seguida por una instrucción else opcional.	
cond statement	Una declaración de cond se usa cuando queremos ejecutar código en base a varias condiciones. Funciona como un if ... else if ... .en muchos otros lenguajes de programación.	

case statement	Se puede considerar como un reemplazo de la declaración switch. Case toma una variable / literal y aplica la coincidencia de patrones con diferentes casos.
	
	Si no se encuentra ninguna coincidencia, sale de la instrucción con un CaseClauseError que muestra que no se encontraron cláusulas coincidentes. Siempre debe tener un caso con _ que coincida con todos los valores. Esto ayuda a prevenir el error mencionado anteriormente. Esto es comparable al caso default.
		``
		case 3 do
		  1 -> IO.puts("Hi, I'm one")
		  2 -> IO.puts("Hi, I'm two")
		  3 -> IO.puts("Hi, I'm three")
		  _ -> IO.puts("Oops, you dont match!")
		end
		``

4. List (Listas)

#Cuando Elixir ve una lista de números ASCII imprimibles los imprime como una lista de caracteres:

```
IO.puts([104, 101, 108, 108, 111])
```

#Dos listas se pueden concatenar y restar usando los operadores ++ y --:

```
IO.puts([1, 2, 3] ++ [4, 5, 6])
IO.puts([1, true, 2, false, 3, true] -- [true, false])
```

#Cabeza y cola de una lista:

```
[head | tail]= [1, 2, 3]
IO.puts(head)
IO.puts(tails)
```

Más funciones de listas:
https://www.tutorialspoint.com/elixir/elixir_lists_and_tuples.htm

5. Tuples (Tuplas)

```
#Definir tuplas
tuple = {:ok, "hello"}

#Longitud de una tupla
IO.puts(tuple_size(tuple))

#Agregar un valor

tuple = Tuple.append(tuple, :world)
IO.puts(tuple_size(tuple))

#Insertar un  valor

tuple = Tuple.insert_at(tuple, 0, :foo)
tuple = put_elem(tuple, 1, :foobar)


#Acceder un elemento de la tupla
IO.puts(elem(tuple, 0))
```

6. Maps (mapas)

Cuando se necesite almacenar un par key-value se usan mapas.

#Crear un mapa: Un mapa es creado usando la sintaxis %{} . Los mapas permiten cualquier valor como clave.

```
map = %{:a => 1, 2 => :b}
```

#Acceder una llave

```
IO.puts(map[:a])
IO.puts(map[2])
```

#Insertar una llave

```
map = Map.put_new(map, :new_val, "value")
IO.puts(map[:new_val])
```

#Actualizar un valor

```
map = %{ map | a: 25}
IO.puts(map[:a])
```

#La coincidencia de patrones

```
%{:a => a} = %{:a => 1, 2 => :b}
IO.puts(a)
```

#Mapas con llave tipo Atom: Cuando todas las claves en un mapa son de tipo atom, puede usar la sintaxis siguiente sintaxis

```
map_dos = %{:a => "Hola", :c => :b}
IO.puts(map_dos.a)
```

7. Modules (módulos)

#Agrupamos una o varias funciones en módulos.

```
defmodule Math do
   def sum(a, b) do
      a + b
   end
end
IO.puts(Math.sum(1, 2))
```

Módulo anidado

```
defmodule Foo do
   #Foo module code here
   defmodule Bar do
      #Bar module code here
   end
end
```

El ejemplo anterior definirá dos módulos: Foo y Foo.Bar.

8. Funciones anónimas:

8.1. Funciones anónimas:

Estas funciones también son llamadas lambdas. Se utilizan asignándoles a variables. Usan las palabras claves `fn` y `end`.

```
sum = fn (a, b) -> a + b end
IO.puts(sum.(1, 5))
```

```
#Usandondo el operador de captura &.
```

```
sum = &(&1 + &2)
IO.puts(sum.(3, 2))
```

#Funciones de coincidencia de patrones:

```
handle_result = fn
   {var1} -> IO.puts("#{var1} found in a tuple!")
   {var_2, var_3} -> IO.puts("#{var_2} and #{var_3} found!")
end
handle_result.({"Hey people"})
handle_result.({"Hello", "World"})
```

Podemos usar la coincidencia de patrones para hacer que nuestras funciones sean polimórficas. Por ejemplo, declararemos una función que puede tomar 1 o 2 entradas (dentro de una tupla) e imprimirlas en la consola:

8.2. Funciones nombradas

Las funciones con nombre se definen dentro de un módulo usando la palabra clave def.

```
defmodule Math do
   def sum(a, b) do
      a + b
   end
end

IO.puts(Math.sum(5, 6))
```

Funciones privadas

Se puede acceder desde el módulo en el que están definidas. Se usa **defp** en lugar de **def**.

```
defmodule Greeter do
    def hello(name), do:
        IO.puts(name)
    defp phrase(), do:
        "Hello "
end
Greeter.hello("world")
```

Si intentamos correr: `Greeter.phrase()` , va a lanzar un error.

Argumentos predeterminados

Si queremos un valor predeterminado para un argumento, usamos la sintaxis de **argumento \\ valor** :

```
defmodule Greeter do
   def hello(name, country \\ "en") do
      phrase(country) <> name)
   end

   defp phrase("en"), do: "Hello, "
   defp phrase("es"), do: "Hola, "
end

Greeter.hello("Ayush", "en")
Greeter.hello("Ayush")
Greeter.hello("Ayush", "es")
```

8.3 Recursiones

```
defmodule Loop do
   def print_multiple_times(msg, n) when n <= 1 do
      IO.puts msg
   end

   def print_multiple_times(msg, n) do
      IO.puts msg
      print_multiple_times(msg, n - 1)
   end
end

Loop.print_multiple_times("Hello", 10)
```

Debido a la inmutabilidad, los bucles en Elixir (como en cualquier lenguaje de programación funcional) se escriben como una recursión: una función se llama de forma recursiva hasta que se alcanza una condición que impida que la acción recursiva continúe.

9. Estructuras:

```
#Definir una estructura
defmodule User do
   defstruct name: "John", age: 27
end


#Las estructuras toman el nombre del módulo en el que están definidas
john = %User{} #john is: %User{age: 27, name: "John"}
ayush = %User{name: "Ayush", age: 20}
```

```
#Para acceder a name y age de john,
IO.puts(john.name)
IO.puts(john.age)

#Para acceder a name y age de ayush,
IO.puts(ayush.name)
IO.puts(ayush.age)

#Y para actualizar el nombre de john
john = %{john | name: "Meg"}
```

10. Procesos:

- En Elixir, todo el código se ejecuta dentro de los procesos.
- Los procesos están aislados unos de otros, se ejecutan simultáneamente y se comunican a través del envío de mensajes.
- Los procesos de Elixir no deben confundirse con los procesos del sistema operativo.
- Los procesos en Elixir son extremadamente livianos en términos de memoria y CPU (a diferencia de los hilos en muchos otros lenguajes de programación). Debido a esto, no es raro tener decenas o incluso cientos de miles de procesos ejecutándose simultáneamente.

10.1. La función Spawn:

La forma más fácil de crear un nuevo proceso es usar la función spawn. Spawn acepta una función que se ejecutará en el nuevo proceso.

```
pid = spawn(fn -> 2 * 2 end)
Process.alive?(pid)
```

ya que los códigos de Elixir se ejecutan dentro de los procesos. Si ejecuta la función self, verá el PID para su sesión actual

```
pid = self
Process.alive?(pid)
```

10.2. Paso de mensajes:

Podemos enviar mensajes a un proceso con **send** y recibirlos con **receive**.

Ejemplo

Pasemos un mensaje al proceso actual y lo recibamos en el mismo.

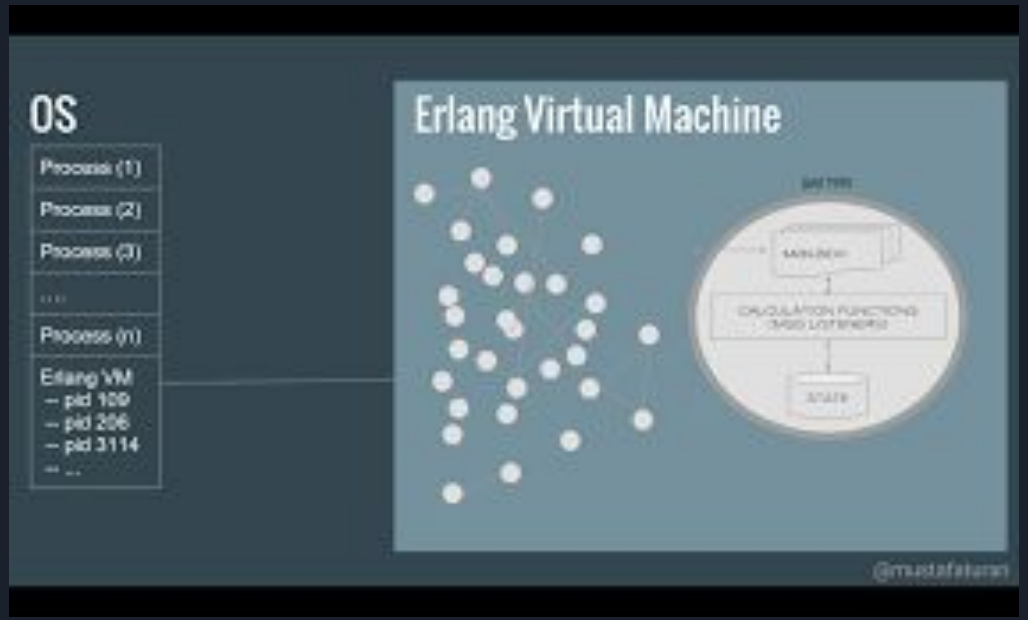
```
send(self(), {:hello, "Hi people"})
receive do
   {:hello, msg} -> IO.puts(msg)
after
   1_000 -> "nothing after 1s"
end
```

Cuando se envía un mensaje a un proceso, el mensaje se almacena en el buzón del proceso. El bloque de recepción pasa por el buzón de proceso actual buscando un mensaje que coincida con cualquiera de los patrones dados.

Si no hay ningún mensaje en el buzón que coincida con ninguno de los patrones, el proceso actual esperará hasta que llegue un mensaje coincidente. También se puede especificar un tiempo de espera con `after`.

CONCURRENCIA EN ELIXIR

Una de las características de Elixir es la manera de realizar concurrencia gracias a la Máquina Virtual de Erlang, Erlang VM (BEAM).



OTP

Acrónimo de Open Telecom Platform. El concepto de OTP está definido por tres componentes principales, definidos por Erlang:

- Erlang
- Librerías disponibles con la Máquina Virtual
- Principios de diseño de sistema (OTP Compliant)

GenServer

Es un bucle que recibe una petición por iteración, junto con un estado de actualización.

En este caso, todas las funciones necesarias se encuentran dentro del paquete `GenServer`.

PROCESOS

Hacen referencia a los procesos de la Máquina Virtual de Erlang, y no del Sistema Operativo. Los procesos de BEAM son más ligeros y menos consumidores que los del S.O., además de que corren en todos los núcleos disponibles.

CREACIÓN DE UN PROCESO EN ELIXIR

La manera más sencilla de crear un nuevo proceso es con `spawn`, pudiendo ser una función anónima o normal:

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

CARACTERÍSTICAS DE LOS PROCESOS DE BEAM

- Están aislados los unos de los otros.
- Se comunican por medio de mensajes.
- Poseen estados explícitos (inmutabilidad).

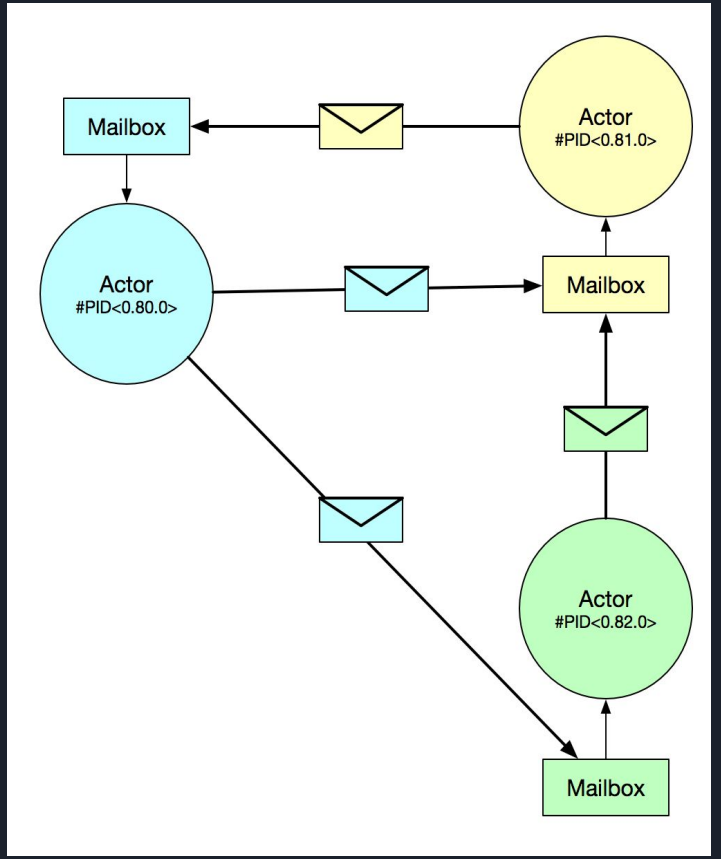
¿CÓMO FUNCIONAN LOS PROCESOS EN ELIXIR?

El funcionamiento de los procesos en Elixir está basado en el Modelo de Actor. En este modelo, un proceso puede guardar su estado, pero este no es compartido

Solo se puede compartir algo entre procesos mediante el envío de mensajes.

Cada proceso posee un "mailbox", el cual se encarga de recibir mensajes de otros procesos.

Al enviar un mensaje, todo sucede de manera asincrónica.



PASO DE MENSAJES EN ELIXIR

Para comunicarse, los procesos en Elixir se basan en el paso de mensajes. Existen dos componentes para hacer esto:

- send
- receive

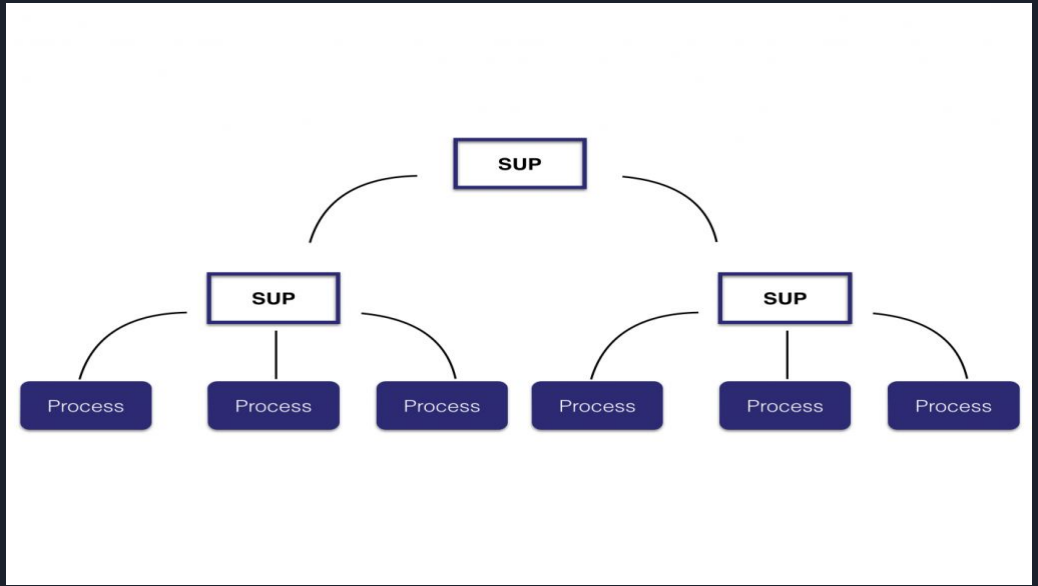
SUPERVISORES

Un proceso puede ser enlazado a otro proceso, lo cual es importante, ya que se pueden detectar fallos y actuar para arreglarlos.

Cuando tenemos procesos que monitorean a otros procesos, los llamamos Supervisores.

SUPERVISION TREE

En caso de tener más de un Supervisor monitoreando los procesos, a esto lo llamamos "Supervision Tree".



ENLACE DE PROCESOS EN ELIXIR

Uno de los problemas de `spawn` es saber cuando un proceso colisiona. Elixir ofrece enlazar los procesos, lo cual hace que reciben una salida de notificación en caso de que ambos sean exitosos, con la siguiente función:

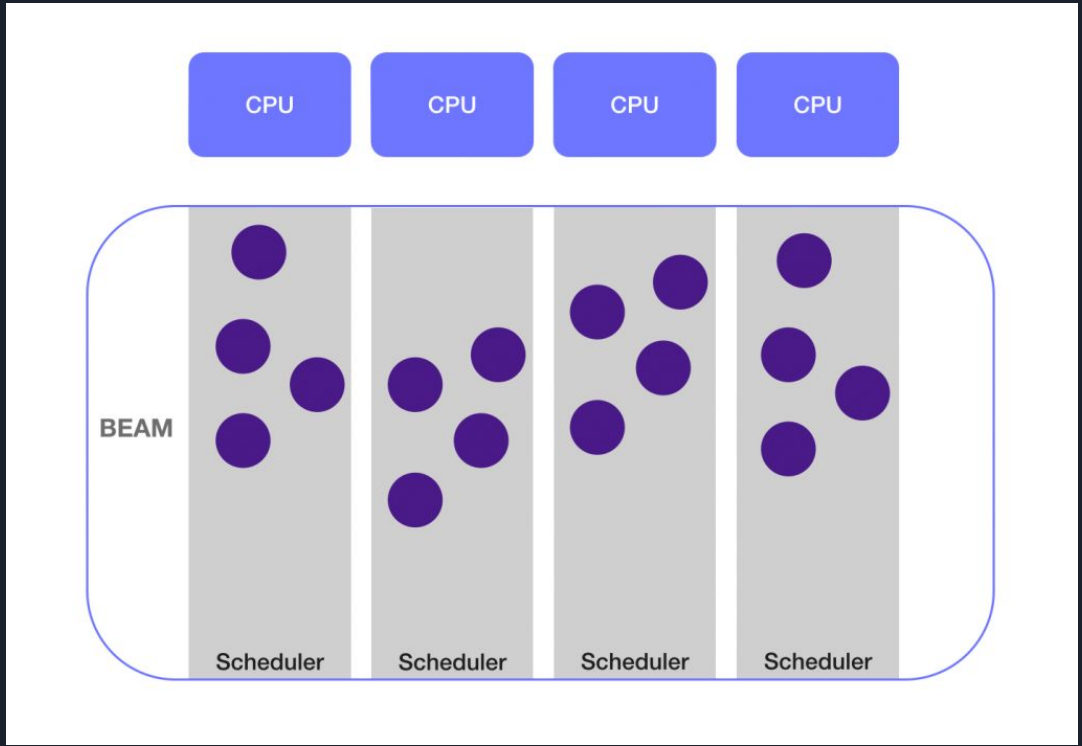
`spawn_link`

CONCURRENCIA

Cuando BEAM se inicia, también lo hace un hilo llamado "Scheduler" encargado de correr cada proceso concurrentemente en la CPU.

Para aprovechar el hardware, BEAM inicia un Scheduler para cada núcleo disponible, con varios procesos corriendo por Scheduler.

CONCURRENCIA



TASKS

Proveen una manera de ejecutar una función de fondo y recuperar su valor luego. Para ello usamos funciones del módulo `Task`, tales como:

- `async`
- `await`

PROS DEL LENGUAJE

Escalabilidad:

Cada proceso de es bastante ligero , lo que permite que miles de procesos corran en la misma maquina, tambien lo hace apto para escalabilidad vertical.

a su vez la facilidad que tienen los procesos de intercomunicarse entre si permite que sea muy fácil la escalabilidad horizontal

Tolerancia a fallos:

los supervisores son una herramienta muy útil al momento de sobrevivir a una falla del sistema ya que ellos poseen instrucciones para el reinicio de ciertas partes del sistema

Velocidad

Los procesos no sólo son ligeros, son también muy rápidos, el tiempo de respuesta de frameworks como ruby on rails se mide en cientos de milisegundos, mientras que en elixir se demoran en el orden de los microsegundos

Adaptabilidad

Elixir nos permite generar software para IoT ya que su velocidad y ligereza ayuda a que los programas para dispositivos con bajos recursos funcionen bien, además de tener un framework que nos ayuda en esas tareas

CONTRAS DEL LENGUAJE

Procesamiento:

Cosas como el manejo de strings y el manejo de datos en bruto no es el fuerte de este lenguaje, hay lenguajes que pueden hacer tareas relacionadas a este tema mucho mejor y con menos recursos, esto obviamente en un nivel de producción, para operaciones normales no presenta un aumento significativo de recursos

Concurrencia =/= paralelismo:

BEAM es un excelente motor para tareas concurrentes y resilientes, sin embargo esto no la hace buena para hacer tareas en paralelo en epecial aquellas que requieran calculos intensivos, para estas tareas se recomienda usar lenguajes especializados como OpenCL

Curva de aprendizaje:

Si bien elixir no es difícil, para hacer un uso completo de él hay que aprender también sobre el paradigma concurrente además de aprender nociones básicas de erlang, BEAM y phoenix, lo cual puede ser abrumador para un recien llegado.

Otro aspecto es que para un recién llegado el código generado en elixir no es fácilmente entendible, por lo que las tareas de mantenimiento pueden resultar caóticas .

Ecosistema:

Como pasa con lenguajes nuevos elixir cuenta con un ecosistema pequeño, hay muchos problemas básicos que no tienen una solución lista para usar, hay 2 opciones para esto: buscar la solución en erlang (por esto es que hay que tener conocimientos en este lenguaje) y la otra es implementar una solución propia.

Con el tiempo este problema ha venido disminuyendo, desde su adopción por grandes compañías Elixir ha ido recibiendo nuevas adiciones por parte de la comunidad.

ALGUNOS MITOS

- La información no es inmutable:
  - VERDADERO Y FALSO: la información es inmutable una vez está en memoria, pero se puede usar rebinding para hacer algo similar a una reasignación de valores

```
iex> animal = "dog"
"dog"
iex> animal = "cat"
"cat"
```

- Los códigos de elixir tienen tendencia a ser un caos
  - FALSO: el código en elixir es sustancialmente diferente para un programador acostumbrado a OOP, y se puede convertir en un caos.

- En elixir no se puede usar el paradigma OO
  - FALSO: este lenguaje no está hecho para esto, pero no significa que no pueda cumplir las tareas de cualquier lenguaje orientado a objetos, no es un reemplazo 1:1 pero `defstruct, defprotocol,defp` y `import` pueden encargarse de hacer definición polimorfismo y encapsulamiento respectivamente
- Al ser inmutable Elixir no tiene estados
  - FALSO: los átomos y los enum pueden representar estados dentro del codigo de ser necesario

CASO DE USO