Guía de Introducción a Elixir

1. Características Claves de Elixir

☑ Funcional

- No hay estado mutable como en lenguajes orientados a objetos.
- Todo se trata como expresiones matemáticas.
- Se basa en funciones puras, lo que facilita el mantenimiento del código.

☑ Concurrencia Nativa

- Usa Procesos Livianos dentro de la BEAM.
- No son threads del sistema operativo.
- Permite que un programa maneje millones de conexiones concurrentes.

☑ Escalabilidad y Distribución

- Puedes correr Elixir en múltiples nodos fácilmente.
- Se diseñó para alta disponibilidad y distribución.

☑ Fácil de Leer y Escribir

- Sintaxis clara y sencilla.
- Inspirado en Ruby, pero sin perder la eficiencia de Erlang.

2. Instalación y Entorno de Desarrollo

- $ sudo apt-get install elixir # En Debian/Ubuntu
- $ elixir -v # Comprobamos que se ha instalado correctamente

3. Trabajando con Elixir

- Para ejecutar comandos en Elixir usar iex, que es su REPL (Read, Eval, Print, Loop).
- Abrir iex en la terminal: $iex
- Para ejecutar operaciones matemáticas:

```
$ iex(1)> 5 + 3
8
$ iex(2)> 10 / 2
5.0
```

4. Tipos de Datos

- Números

- $ iex> 10 # Enteros
- $ iex> 3.14 # Flotantes
- $ iex> 0x1F # Hexadecimales

- Átomos (:atom)

- $ iex> :ok
- :ok
- $ iex> :error
- :error
- $ iex> :hello
- :hello

- Cadenas (String); Las cadenas son binaries en UTF-8.

- $ iex> "Hola mundo"
- "Hola mundo"
- $ iex> "Elixir" <> " Rocks!"
- "Elixir Rocks!"

- Listas ([ ])

- $ iex> [1, 2, 3]
- [1, 2, 3]
- $ iex> [1 | [2, 3]]
- [1, 2, 3]
- $ iex> length([1,2,3])
- 3

- Tuplas ({ })

- $ iex> {1, 2, 3}
- {1, 2, 3}
- $ iex> {:ok, "Todo bien"}
- {:ok, "Todo bien"}

5. Variables y Pattern Matching

- Las variables en Elixir no almacenan valores, sino que se enlazan a ellos.

```
$ iex> x = 5
5
$ iex> y = 10
10
$ iex> x + y
15
```

⚠️ Las variables son inmutables:

```
$ iex> x = 5
$ iex> x = 10  # Esto no re-asigna x, simplemente la enlaza a un nuevo valor.
```

⚠️ Pattern Matching. ES UNA TÉCNICA CLAVE DE ELIXIR.

```
$ iex> {a, b} = {1, 2}
$ iex> a
1
$ iex> b
2
```

⚠️ Si el patrón no coincide, lanza un error:

```
$ iex> {x, y} = {1, 2, 3}  # Error, porque la tupla no coincide.
```

6. Funciones en Elixir

- Las funciones se definen dentro de módulos.

Definiendo un módulo con funciones:

```
defmodule Matematica do
 def suma(a, b) do
  a + b
 end

 def resta(a, b) do
  a - b
 end
end
```

Llamando funciones:

```
$ iex> Matematica.suma(5, 3)
8
```

Funciones Anónimas:

```
$ iex> suma = fn a, b -> a + b end
$ iex> suma.(2, 3)
5
```

7. Pipe Operator (|>)

- Permite encadenar funciones sin necesidad de anidarlas.

```
$ iex> "elixir" |> String.upcase() |> String.reverse()
"RIXILE"
```

8. Mix: Herramienta para Proyectos

- mix es la herramienta para crear y gestionar proyectos en Elixir.

⚠️ Crear un nuevo proyecto: $ mix new mi_proyecto

Esto generará una estructura con archivos y carpetas necesarias.

Ejercicios

Ejercicio 7: match_123

- Crear una función match_123/1 que reciba una lista y haga pattern matching para devolver los elementos después de [1,2,3].

```
defmodule Sheet1 do
 def match_123([1, 2, 3 | tail]) do
  tail
 end
end
```

Ejercicio 8: match_string

- Crear una función que haga pattern matching con strings.

```
defmodule Sheet1 do
 def match_string("Hello " <> name) do
  name
 end
end
```

Ejercicio 9: match_1234

- Modificar match_123 para que use el operador ++.

```
defmodule Sheet1 do
 def match_1234([1, 2, 3] ++ tail) do
  tail
 end
end
```

🔍 Pruebas Automáticas

- Crea el archivo test/sheet1_test.exs con:

```
defmodule Sheet1Test do
 use ExUnit.Case
 doctest Sheet1

 test "match_123 works correctly" do
  assert Sheet1.match_123([1, 2, 3, 4, 5]) == [4, 5]
 end
end
```