Due Process

elixir

"Elixir is a dynamic, functional language for building scalable and maintainable applications."

Basic Data Types

```
# Comments start with #
1 # Integer
2.3 # Float
"Hello" # Binary (String)
{1, 2, 3} # Tuple
[1, 2, 3] # List
:ok # Atom
```

Modules and Functions

```elixir
defmodule Example do
  def say_hello do
    "Hello, World!"
  end

  def say_howdy, do: "Howdy, World!"
end

Example.say_hello() # => "Hello, World!"
Example.say_howdy() # => "Howdy, World!"

double = fn (x) -> x * 2 end
double.(4) # => 8
```

Pattern Matching

```
{a, b, c} = {1, 2, 3}
a # => 1
b # => 2

{1, x, 3} = {1, 2, 3}
x # => 2

{1, x, 3} = {1, 2, 4}
# => ** (MatchError) no match of right hand side value: {1, 2, 4}
```

Pattern Matching Cont.

```
defmodule Example do
  def say_hello("Alice") do
    "Howdy, Alice!"
  end

  def say_hello("Bob") do
    "Yo, Bob!"
  end

  def say_hello(name) when is_binary(name) do
    "Hello, #{name}!"
  end
end
```

Pattern Matching Cont.

```elixir
defmodule Example do
  def say_hello("Alice") do
    "Howdy, Alice!"
  end

  def say_hello("Bob") do
    "Yo, Bob!"
  end

  def say_hello(name) when is_binary(name) do
    "Hello, #{name}!"
  end
end
```

```elixir
Example.say_hello("Alice") # => "Howdy, Alice!"
Example.say_hello("Bob") # => "Yo, Bob!"
Example.say_hello("Carol") # => "Hello, Carol!"
Example.say_hello(%{ key: 1 })
# => ** (FunctionClauseError) no function clause matching in Example.say_hello/1
```

Recursion

```
defmodule Example do
  def last([]), do: nil
  def last([elem]), do: elem
  def last([elem | rest]), do: last(rest)
end
```

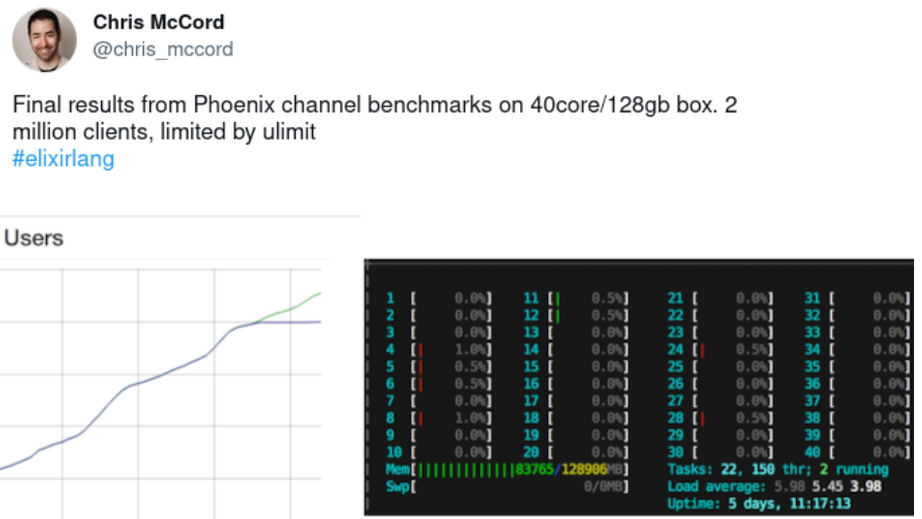
Recursion

```elixir
defmodule Example do
  def last([]), do: nil
  def last([elem]), do: elem
  def last([elem | rest]), do: last(rest)
end
```

```elixir
Example.last([]) # => nil
Example.last([1]) # => 1
Example.last([1, 2, 3]) # => 3
Example.last(Enum.to_list(1..999_999)) # => 999999
```

Processes

- "Soul" of Elixir & Erlang
- Actor Model
- Map transparently to hardware



Basic Process

```
pid = spawn(fn () ->
  receive do
    message -> IO.puts("Received #{message}")
  end
end)
```

Basic Process

```elixir
pid = spawn(fn () ->
  receive do
    message -> IO.puts("Received #{message}")
  end
end)
```

```elixir
Process.alive?(pid) # => true
send(pid, "first") # => Received first
send(pid, "second") # => ...Nothing?
Process.alive?(pid) # => false
```

Keeping It Alive

```
defmodule Example do
  def server do
    receive do
      {:echo, message} ->
        IO.puts("Server received #{message}")
        server()
      :kill ->
        IO.puts("Goodbye!")
    end
  end
end
```

Keeping It Alive

```elixir
defmodule Example do
  def server do
    receive do
      {:echo, message} ->
        IO.puts("Server received #{message}")
        server()
      :kill ->
        IO.puts("Goodbye!")
    end
  end
end
```

```elixir
pid = spawn(Example, :server, [])
Process.alive?(pid) # => true

send(pid, {:echo, "first"})  # => Server received first
send(pid, {:echo, "second"})  # => Server received second
Process.alive?(pid) # => true

send(pid, :kill) # => Goodbye!
Process.alive?(pid) # => false
```

Process State

```
defmodule Counter do
  def server(current_amount \\ 0) do
    receive do
      {:add, amount_to_add} ->
        server(current_amount + amount_to_add)
      {:get, pid} ->
        send(pid, current_amount)
        server(current_amount)
    end
  end
end
```

Process State Cont.

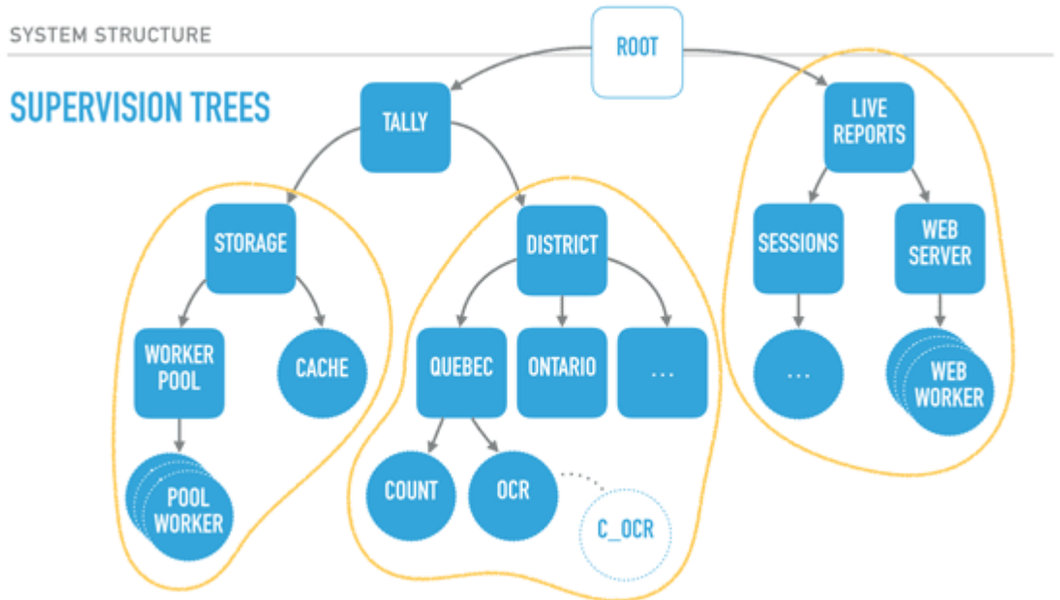
```elixir
server_pid = spawn(Counter, :server, [])

send(server_pid, {:add, 5})
send(server_pid, {:get, self()})
receive do
  amount -> IO.puts("Received #{amount}")
end
# => Received 5

send(server_pid, {:add, 2})
send(server_pid, {:get, self()})
receive do
  amount -> IO.puts("Received #{amount}")
end
# => Received 7
```

Open Telecom Platform (OTP)

- Standardizes spawning and monitoring of processes
- Fire barriers within your app
- "Let it crash"



GenServer (Generic Server)

```elixir
defmodule Stack do
  use GenServer

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

  def handle_call(:pop, _from, [head | tail]) do
    {:reply, head, tail}
  end

  def handle_cast({:push, element}, state) do
    {:noreply, [element | state]}
  end
end
```

GenServer (Generic Server)

```elixir
defmodule Stack do
  use GenServer

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

  def handle_call(:pop, _from, [head | tail]) do
    {:reply, head, tail}
  end

  def handle_cast({:push, element}, state) do
    {:noreply, [element | state]}
  end
end
```

```elixir
{:ok, pid} = GenServer.start_link(Stack, [:first_elem])
GenServer.cast(pid, {:push, :second_elem})
GenServer.call(pid, :pop) # => :second_elem
GenServer.call(pid, :pop) # => :first_elem
```

Better GenServer

```
defmodule Stack do
  use GenServer

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

  def push(pid, element) do
    GenServer.cast(pid, {:push, element})
  end

  def pop(pid) do
    GenServer.call(pid, :pop)
  end

  #
```

Better GenServer

```elixir
defmodule Stack do
  use GenServer

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

  def push(pid, element) do
    GenServer.cast(pid, {:push, element})
  end

  def pop(pid) do
    GenServer.call(pid, :pop)
  end

  #
```

```elixir
{:ok, pid} = Stack.start_link()
Stack.push(pid, :something)
Stack.pop(pid) # => :something
```

And Much More

- Agent
- Supervisor
- Task
- Registry
- Stream

Learn More

- Elixir: The Documentary
- Elixir Guide
- The Zen of Erlang
- A Week with Elixir