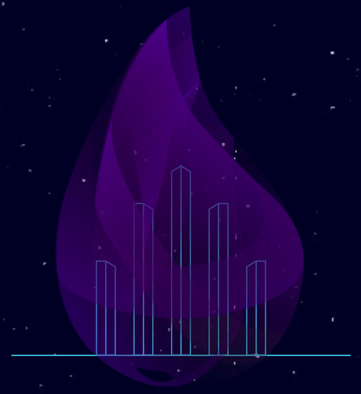
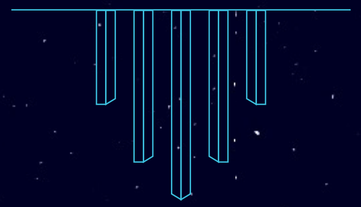
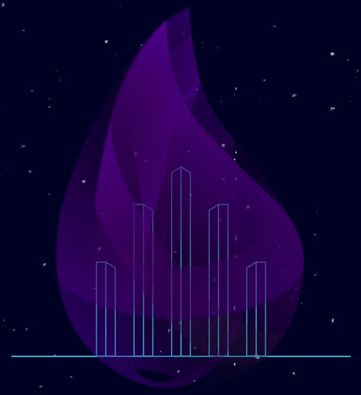


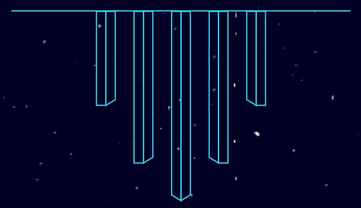
ELIXIR

Functional, Concurrent, Distributed



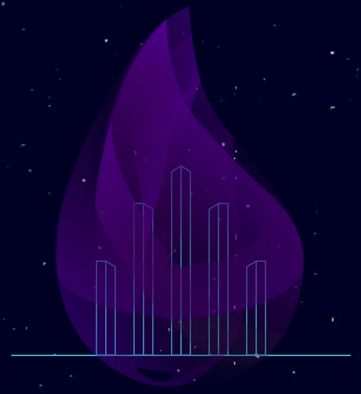


PRACTICAL REALITY

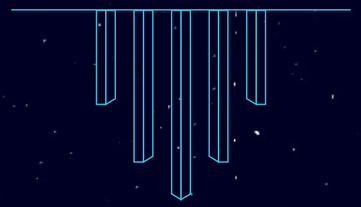


Practical Reality

- Can’t replace PHP/Python any time soon (don’t worry Simon)
- Basically impossible to find Elixir/Erlang developers
- But Elixir is perfect for apps that require real-time communication or high levels of concurrency
- Learning another programing paradigm can only help us become better developers



HISTORY



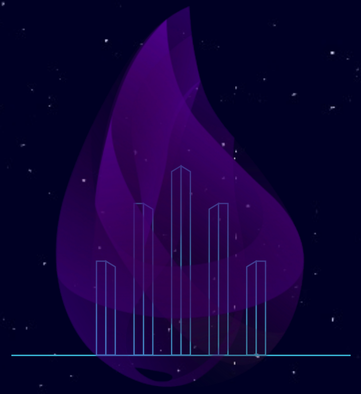
Erlang

- Functional, dynamically typed language created by Ericsson in the 1980's
- Designed for telecom applications
- fault tolerant, scalable and distributed systems

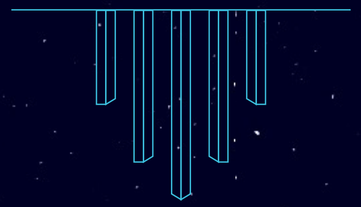
- In 1998 Ericsson proved that telecom switch running Erlang achieved nine "9"s of availability
- Used by WhatsApp, Heroku, Amazon, Facebook, Discord, etc

LET IT CRASH

- Don’t rescue exceptions
- Don't clutter the "happy path.”
- Let the program crash and allow supervisors to put everything back together
- Everything is supervised → fault tolerant systems

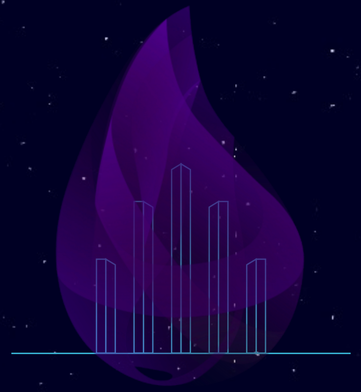


OTP

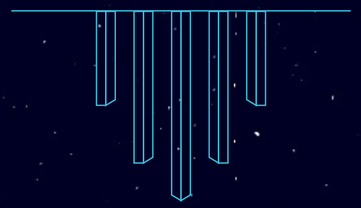


Open Telecom Platform

- Always use Supervision Trees
  - structuring model based on the idea of workers and supervisors:
- Standard library provides Generic Behaviors
  - common patterns abstracted to a generic behavior and a specific callback
    - e.g. GenServer, Generic Finite State Machine, Generic Event Handle, In Memory Storage, etc
- Everything is an Application
  - encapsulate resources and functionality



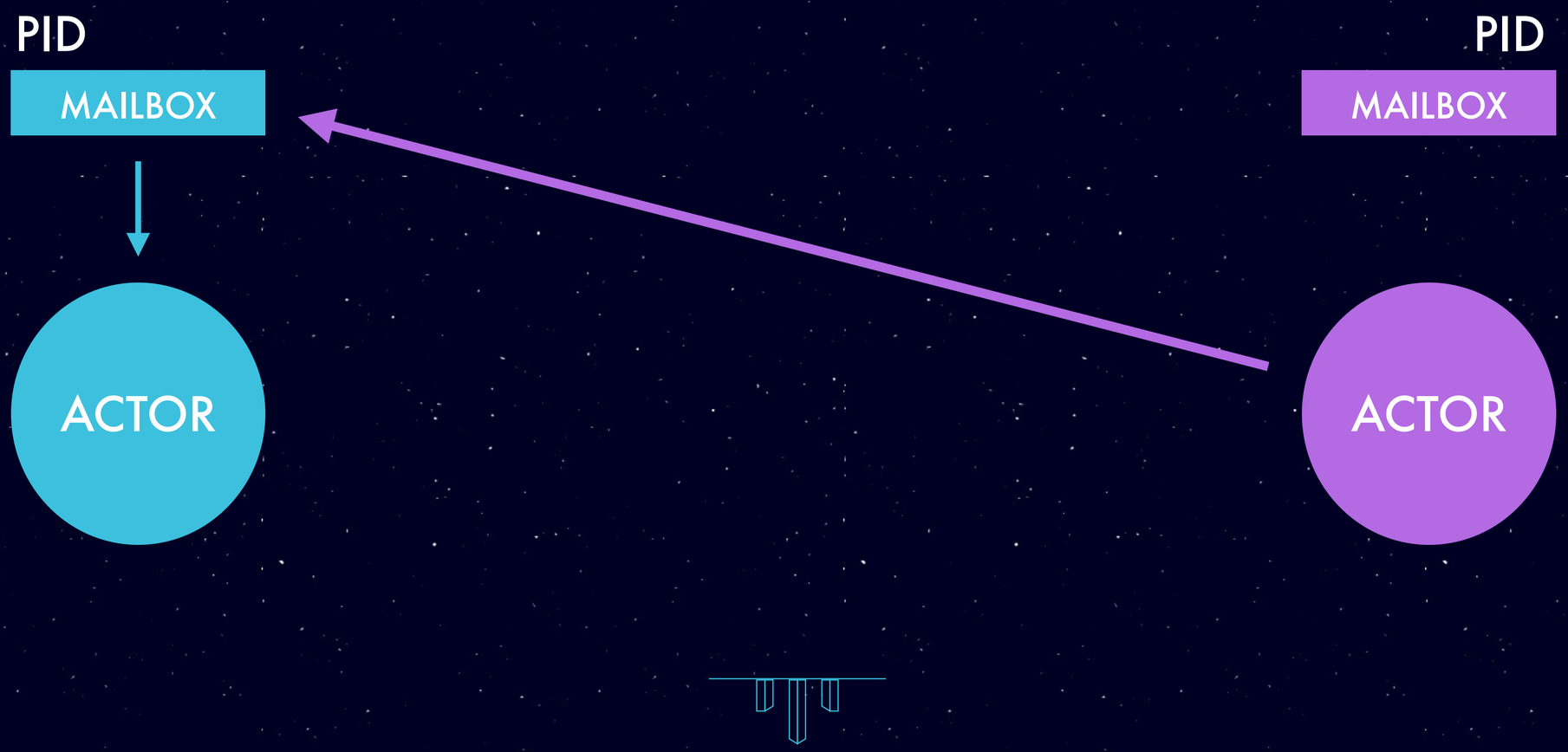
ACTOR MODEL



Actor Model in Erlang/Elixir

- A model for distributed and concurrent computation
- Everything is an Actor
- Actors have their own memory and are completely isolated
- Actors communicate asynchronously with other actors
- Actors process one message at a time and respond in 3 ways
  - Send messages
  - Create new Actors
  - Change internal state

Actor Model in Erlang/Elixir

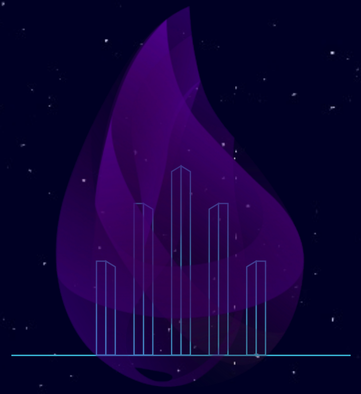


Actor Model in Erlang/Elixir

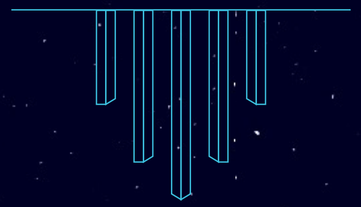
- In Elixir/Erlang Actors are referred to as "`process`"
- Each `process` has a PID address
- `process` run an endless recession loop to keep state
- Everything is monitored by a Supervisor (just another `process`)

Threads

- “But wait, I can already do asynchronous programing just fine with threads, why do I care about another model?”
- Object communicate synchronously
- How do we avoid corrupting the internal state of an object
- Thread Locking



RISE of ELIXIR



Rise of Elixir

- José Valim, a core-contributor to Rails was frustrated with Ruby's inability to handle concurrency
- In 2011, Inspired by Erlang, Valim decided to create a "Ruby flavor" designed to have better syntax, tooling, and documentation
- Elixir uses the same abstractions and re-uses the good parts of Erlang
- All Elixir is compiled to Erlang byte code and runs in the Erlang VM (BEAM)

Elixir Basics

- Functional
- Immutable
- Pattern Matching
- Piping
- Processes
- Testing
- Dependencies & Documentation
- Access to all of Erlang

Functional Programing

- “A programming paradigm that treats computation as the evaluation of pure mathematical functions and avoids changing-state, side-effects, and mutable data.”

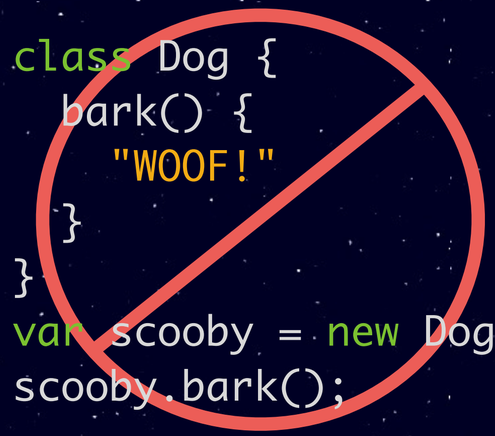
Object Oriented Design?

- “Paradigm where functionality is organized using ‘objects’ which contain data (attributes) and procedures (methods) and are self-referential”

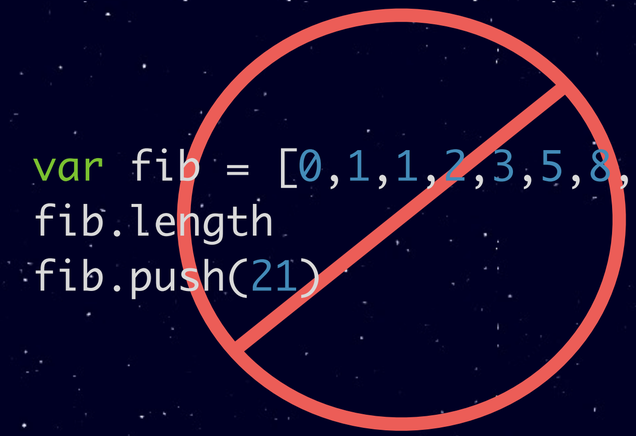
```
class Dog {
  constructor(name) {
    this.name = name;
  }
  bark() {
    return `${name} barks WOOF!`
  }
}
var scooby = new Dog("scobby");
scooby.bark()
```

Functional

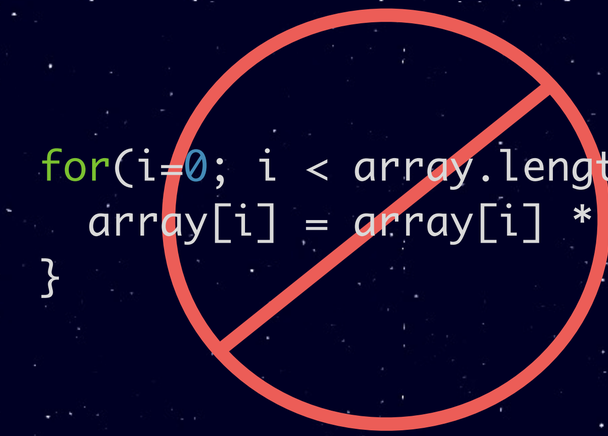
- No Classes, No Objects, No for, while, do loops



```
class Dog {
  bark() {
    "WOOF!"
  }
}
var scooby = new Dog();
scooby.bark();
```



```
var fib = [0,1,1,2,3,5,8,13]
fib.length
fib.push(21)
```



```
for(i=0; i < array.length; i++) {
  array[i] = array[i] * 2;
}
```

FUNCTIONAL

- Modules, Processes, Recursion

```
defmodule Dog do
   defstruct name: nil

   def bark(name) do
      "#{name} barks WOOF!"
   end
end
scooby = %Dog{
           name: “scooby”
         }
Dog.bark(scooby.name)
```

```
def fib() do
    fib = [0,1,1,2,3,5,8,13]
    Enum.count(fib)
    fib ++ [21]
end
```

```
def multiply([]) do
    []
end
def mutiply(array) do
    [head | tail] = array
    [head * 2] ++ mutiply(tail)
end
```

Immutable

- Never mutate state. Return a new copy every time. But can reset references

```
tuple = {:ok, "Elliot"}
>> {:ok, "Elliot"}
put_elem(tuple, 1, "Mr. Robot")
>> {:ok, "Mr. Robot"}
tuple
>> {:ok, "Elliot"}
```

Pattern Matching

- Deconstruct data structures with the “match operator”

```
def deconstruction() do
   {:ok, direction} =  Map.fetch(@data, :a)
   "We are ready to go {direction}"
end
```

Pattern Matching

- The world is your switch statement

```
def matching_on_args("Is it weakness?") do
   "Avoid the blind woman"
end
def matching_on_args("Is it wickedness?") do
   "You've lost...your life."
end
def matching_on_args(_) do
   "You decide. Are we gonna live or die?"
end
```

Pattern Matching

- Conditional logic by de-structuring data

```
def conditional() do
    case Map.fetch(@data, :a) do
      {:ok, "left"} -> "My left stroke just went viral"
      {:ok, _} -> "Right stroke put lil' baby in a spiral"
      {:error} -> "Sit down"
    end
end
```

Pipe Operator

- Functional programing is all about chaining

```
def traditional() do
    String.reverse(
      :zlib.uncompress(
        Base.decode64!(
          "eJzLy88sKchXqMzJy1dIzShRKFYvSQR=="
        )
      )
    )
end
```

Pipe Operator

- Elixir makes it very attractive to chain

```elixir
def chain() do
    "eJzLy88sKchXqMzJy1dIzShRKFYvSQR=="
    |> Base.decode64!
    |> :zlib.uncompress
    |> String.reverse
end
```

Processes (Actors in Elixir)

- Everything is a `process`
- `process` are completely isolated.
- `process` creation and destruction is super lightweight
- Message passing is the only way for `processes` to interact
- `processes` have unique names (PID)
- You can send any `process` a message
- `processes` do what they are supposed to do or fail

Processes

- Spawn a simple `process`

```
defmodule Example do
  def yah do
    receive do
      :yah -> IO.inspect("yah, yah")
    end
  end
end
```

```
pid = spawn(Example, :yah, [])
send(pid, :yah)
"yah, yah"
```

Processes

- Help `processes` have a conversation

```
defmodule Process do
  def yah_yah do
    receive do
      {sender , msg} ->
       send(sender, {:ok, "#{msg}, yah, yah"})
      yah_yah()
    end
  end

  def client do
    receive do
      {:ok, msg} -> IO.inspect(msg)
      client()
    end
  end
end
```

Processes

- Help `processes` have a conversation

```
pid_echo = spawn(Process, :yah_yah, [])
pid_client = spawn(Process, :client, [])

send(pid_echo, {pid_client, "Zeroes to flip, temptation is"})
"Zeroes to flip, temptation is, yah, yah"

send(pid_echo, {pid_client, "Buzzin', radars is buzzin'"})
"Buzzin', radars is buzzin', yah, yah"
```

Processes

- Stack Overflow? → Elixir uses “tail-call” optimization

```
defmodule Process do
  def yah_yah do
    receive do
      {sender , msg} ->
        send(sender, {:ok, "#{msg}, yah, yah"})
        yah_yah()
    end
  end

  def client do
    receive do
      {:ok, msg} -> IO.inspect(msg)
      client()
    end
  end
end
```

```
pid_echo = spawn(Process, :yah_yah, [])
pid_client = spawn(Process, :client, [])

send(pid_echo, {pid_client, "Zeroes to flip, temptation is"})
"Zeroes to flip, temptation is, yah, yah"

send(pid_echo, {pid_client, "Buzzin', radars is buzzin'"})
"Buzzin', radars is buzzin', yah, yah"
```

Testing

- Mix, ExUnit, Doc Tests

```
test "can mutiply array elements by 2" do
    assert mutiply_by_two([10,11]) == [20,22]
end
```

```
@doc ~S"""
Parses the given `line` into a command.

## Examples

    iex> Example.Functional.mutiply_by_two([1,2,3,4])
    [2,4,6,8]

"""
def mutiply_by_two([]) do
    []
end
def mutiply_by_two(array) do
    [head | tail] = array
    [head * 2] ++ mutiply_by_two(tail)
end
```

Documentation & Dependencies

- Mix
  - dependency management and app management all wrapped into a standard library
  - Can build single binary releases
- Hex
  - Community of open source libraries with auto generated documentation

All of Erlang

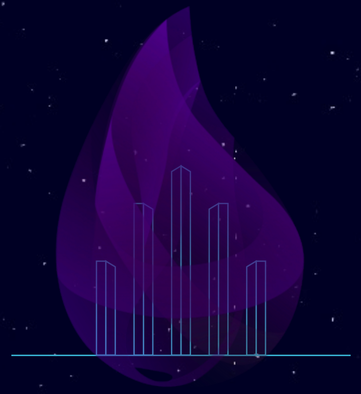
- The power of 35+ years of Erlang is in your hands

```
:ets.insert(table, {"E-Corp", 322_000_000})

:math.pi()

:rand.uniform()

:crypto.hash(:sha, "LEAVE ME HERE")
```

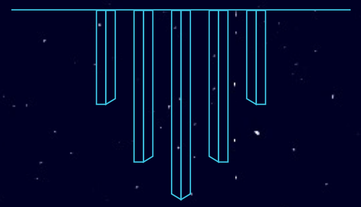
BENEFITS

Benefits

- Concurrency, Concurrency, Concurrency + Fault Tolerance
- Performance (better than PHP/Python/Ruby/Node but slower than Go)
- Patterns that are better suited for web development (same challenges as telecom)
- Modern language with first class support for tooling (dependencies, testing, documentation)

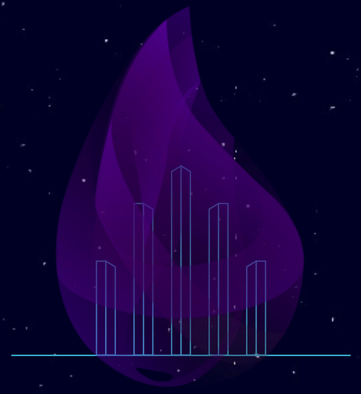


DRAWBACKS

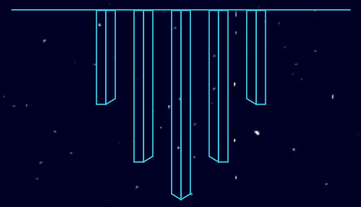


Drawbacks

- Very different from most of the languages and patterns we use
- Mature infrastructure that has grown in parallel to most other web languages
- Young language with small community
- Not well suited for computation intensive tasks (Big Data, Machine Learning, etc)



EXAMPLES



Using The Actor Model

- Keep state in a **`process`**
- **`process`** runs in an endless recursion
- Use Agent OTP library to reduce boilerplate

```
defmodule Box do

  def start_link(color) do
    Agent.start_link(fn -> [] end, name: color)
  end

  def get(box) do
    Agent.get(box, fn list -> list end)
  end

  def push(box, value) do
    Agent.update(box, fn list -> [value|list] end)
  end

end
```

Processes Communication

- How to manage **process** communication

```elixir
defmodule Portal do

  def transfer(left, right, data) do
    data
    |> Enum.each(fn item -> Box.push(left, item) end)
  end

  def push_right(portal) do
    case Box.pop(portal.left) do
      :error    -> :ok
      {:ok, head} -> Box.push(portal.right, head)
    end
    portal
  end

end
```

Processes Communication

- Transfer data between Boxes (**`processes`**)

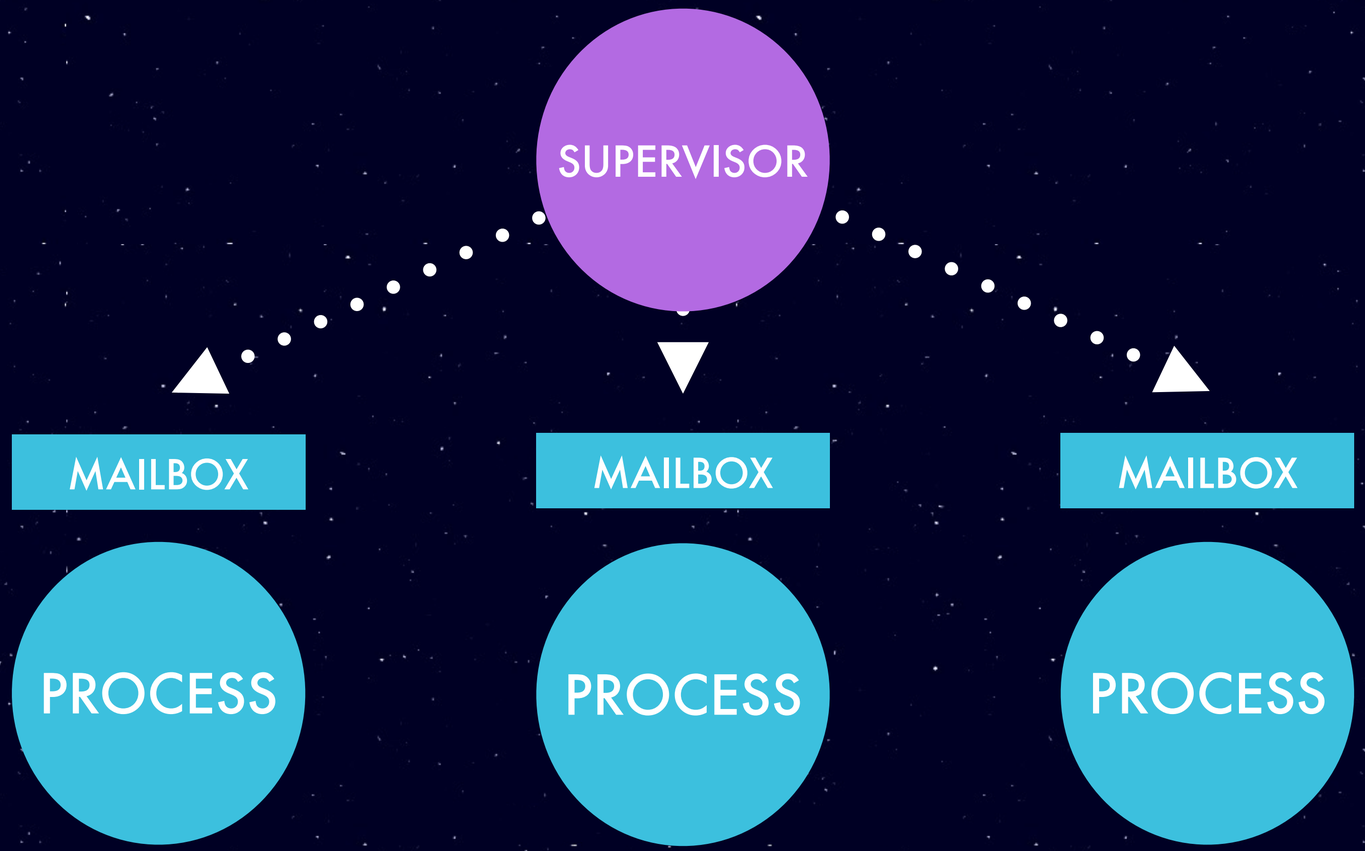
```
Box.start_link(:orange)
Box.start_link(:blue)

portal = Portal.transfer(:orange, :blue, [1, 2, 3])

Box.get(:orange) # => [3, 2, 1]
Portal.push_right(portal)
Box.get(:orange) # => [2, 1]
```

Actor Communication

- But what if one of the **`processes`** dies?

```
Process.unlink(Process.whereis(:blue))

Process.exit(Process.whereis(:blue), :shutdown)

Portal.push_right(portal)
# => ** (EXIT) no process **
```

Fault Tolerance



Fault Tolerance

- Follow OTP and add a Supervisor

```
defmodule Fuzzional.Application do
  use Application

  def start(_type, _args) do
    import Supervisor.Spec, warn: false

    children = [ worker(Fuzzional.Box, []) ]

    opts = [strategy: :simple_one_for_one, name: Fuzzional.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Faul Tolerant

- Supervisor will automatically create a new **`process`** to match

```
Box.start_link(:orange)
Supervisor.start_child(Fuzzional.Supervisor, [:blue])

portal = Portal.transfer(:orange, :blue, [1,2,3])

Process.unlink(Process.whereis(:blue))
Process.exit(Process.whereis(:blue), :shutdown)

Portal.push_right(portal)
assert Box.get(:blue) # => [3]
```

Distributed

- Because **`processes`** are completely isolated they can easily communicate across a network
- OTP Agent library is can handle cross-node communication all we need to do is pass the node name

```
{:blue, “node@aws"} {:orange, "node@azure"}
```

- Distributed transfers without changing a single line of code!

Code Fishbowl: http://104.131.16.3/

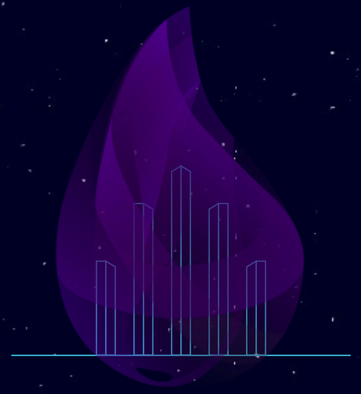
- Building real-time applications
- Phoenix
- Web sockets handled by `processes`
- Using Erlang's built in NoSQL DB
- Single binary deployment

The venerable master Brez was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said “Master Brez, I have heard that objects are a very good thing - is this true?" Brez looked pityingly at his student and replied, "Foolish pupil - objects are merely a poor man’s closures.”

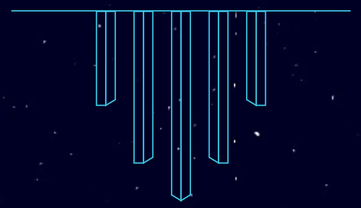
Chastised, Anton took his leave intent on studying closures. He carefully read the entire “Lambda Calculus" series of papers, and implemented a small Scheme interpreter with a closure-based object system. He learned much, and looked forward to informing master Brez of his progress.

On his next walk with Brez, Anton attempted to impress the master by saying “Master Brez, I have diligently studied the matter, and now understand that objects are truly a poor man's closures." Brez responded by hitting Anton with his stick, saying "When will you learn? Closures are a poor man’s object."

At that moment, Anton became enlightened.



QUESTIONS?



RESOURCES

- http://elixir-lang.org/
- https://elixirschool.com/
- http://howistart.org/posts/elixir/1/
- https://media.pragprog.com/titles/elixir/ElixirCheat.pdf
- https://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/
- https://medium.com/@cscalfani/so-you-want-to-be-a-functional-programmer-part-1-1f15e387e536