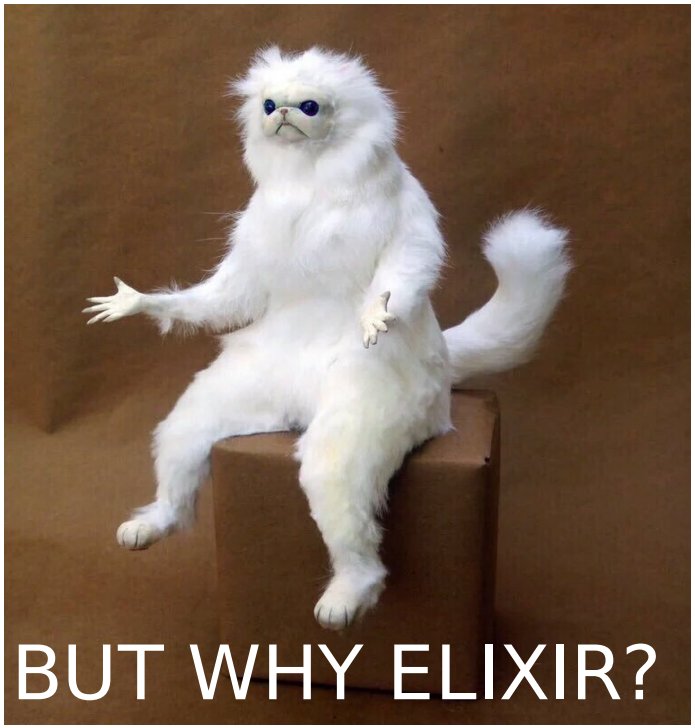
DRINKING SOME ELIXIR

WHAT IS ELIXIR ?

Elixir is a functional, concurrent, general-purpose programming language that runs on the BEAM VM.



WHY ELIXIR? (SCALABLE)

Most JS/Python/Ruby Apps

Erlang/Elixir Apps

WHY ELIXIR? (RESPONSIVE AND ROBUST)

- Runs on Erlang VM
  - Fault Tolerant
  - Distributed
  - Low latency
  - Consistent performance
  - Observable

*We will see these in code later

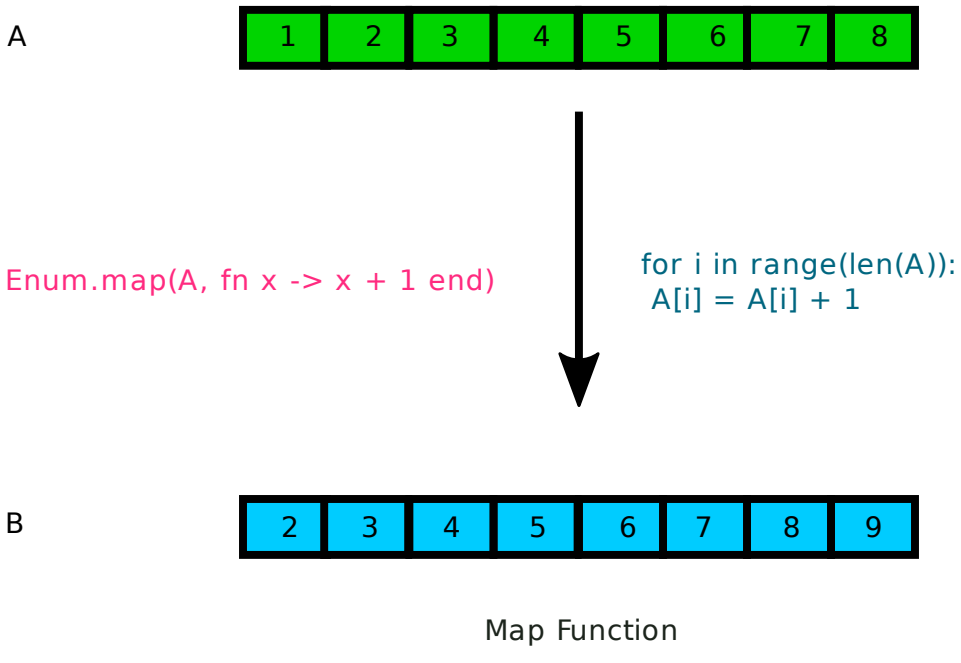
WHY ELIXIR? (PROGRAMMER INCENTIVES)

- Mature Tooling and Documentation
- Use Erlang libraries
- Metaprogramming
- Standards for distributed apps (OTP)
- Actor model based concurrency
- Functional (is the new cool :D)

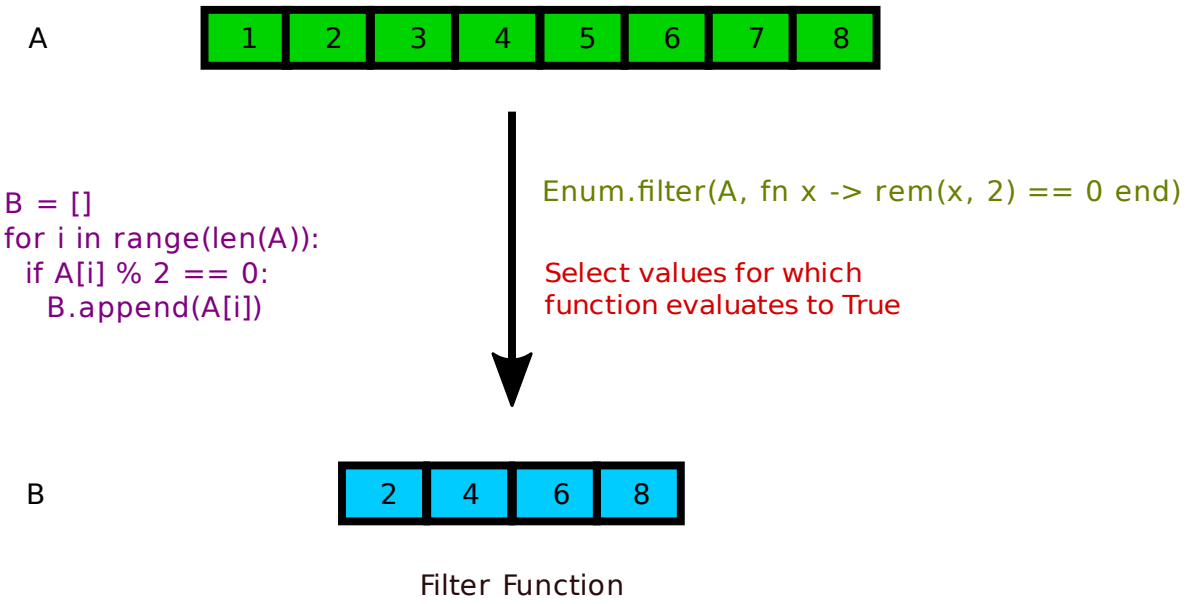
A VERY SHORT INTRO TO FP

- Data is immutable
- Functions are data transformers
- Recursion favored over loops
- Functions can take functions as arguments

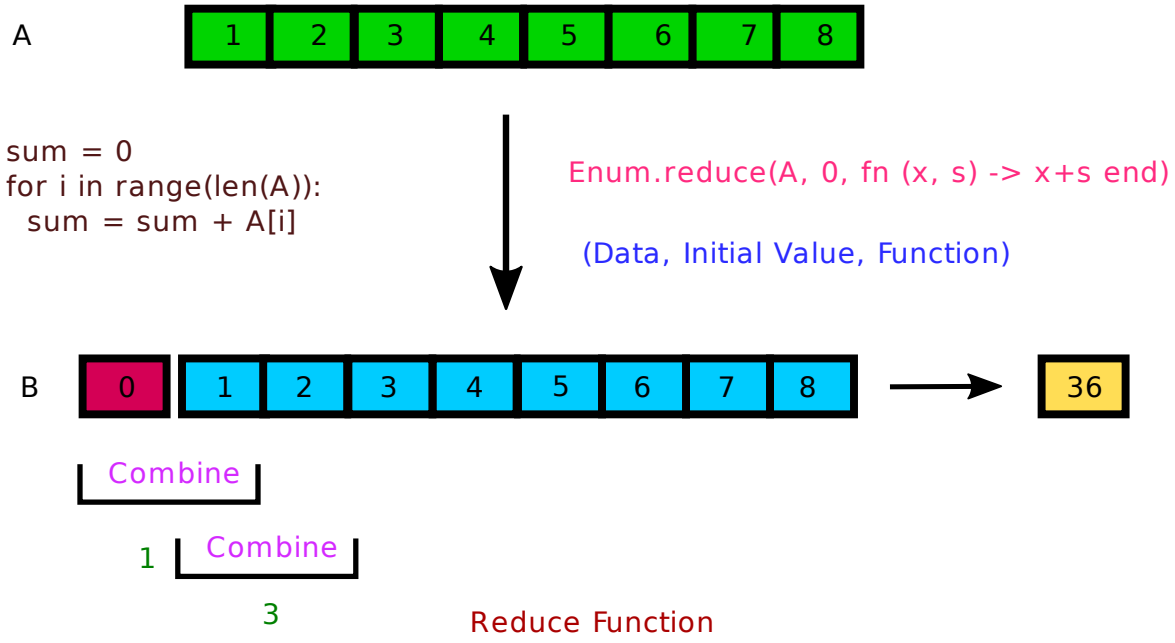
MAP OVER DATA



FILTER OVER DATA



REDUCE OVER DATA

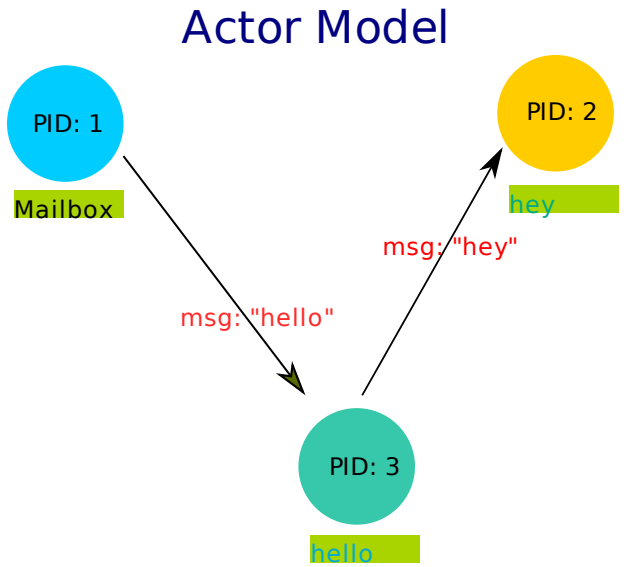


BASIC CONCEPTS

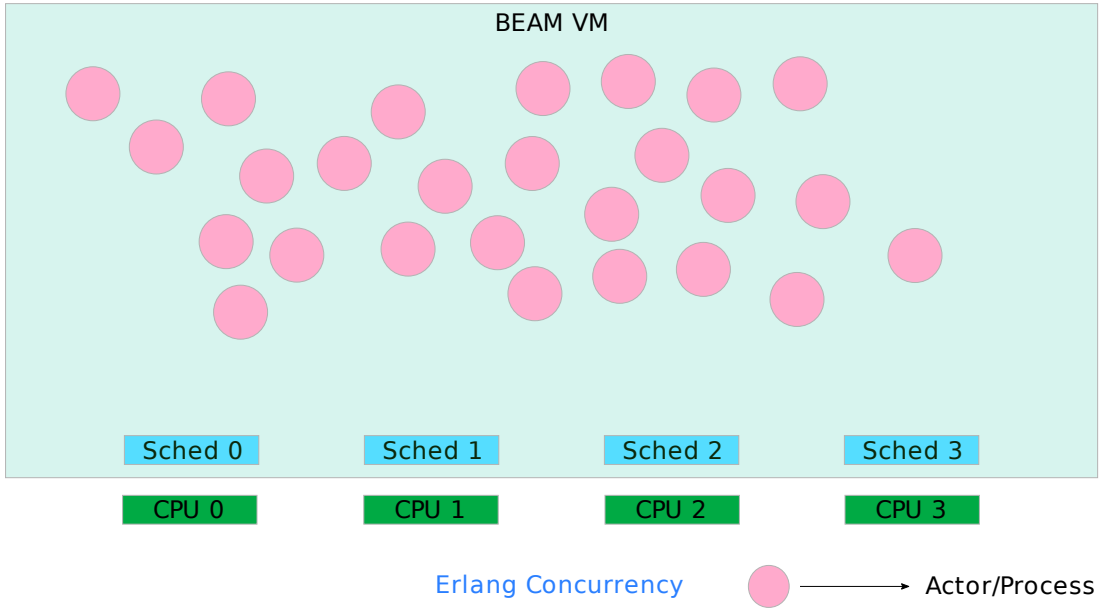
- Actors
- Links/Monitors
- Erlang VM Overview
- Supervisors (OTP)
- Application (OTP)
- Releases (OTP)

ACTORS

- Entities which, communicate with message passing
- Send/Receive message (asynchronous)
- Location/Network transparent
- All code runs inside an actor/process
- Building block for concurrency and distribution



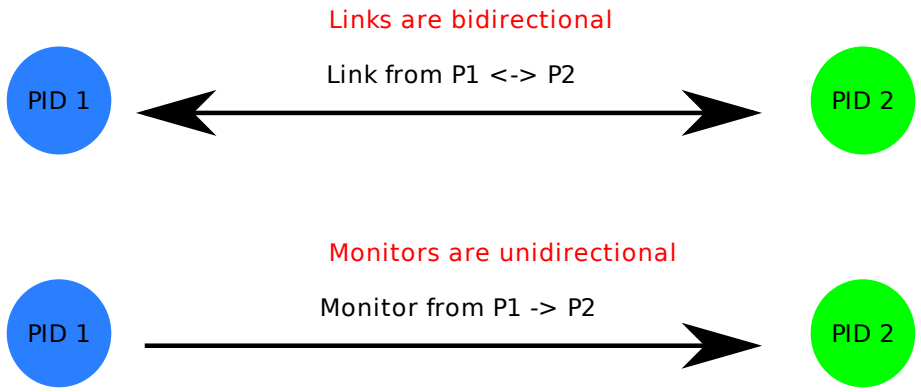
ERLANG VM OVERVIEW



- A Preemptive scheduler for each CPU
- Extremely lightweight processes
- Process level Garbage Collection
- Asynchronous message passing
- Can handle millions of processes

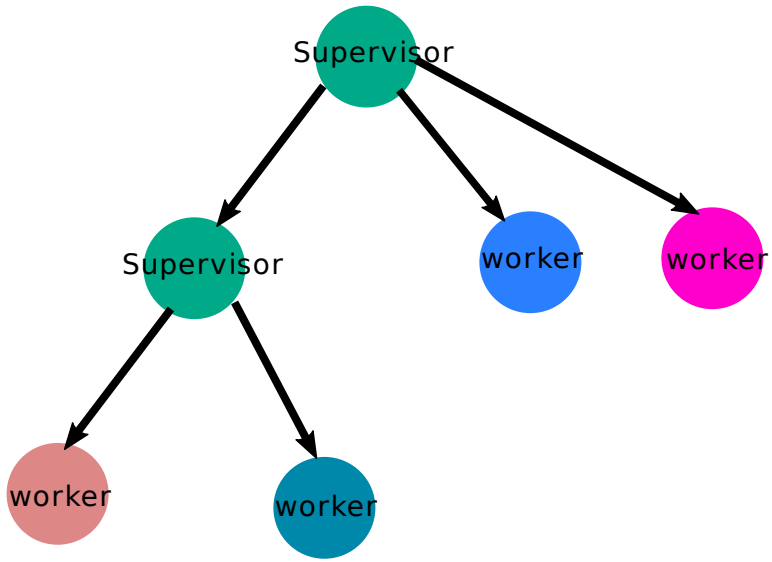
LINKS AND MONITORS

- Link - Links one process to another (Not stackable)
- Monitor - Monitors a process (Stackable)
- Gets notifications on linked/monitored processes
- Pillars for building fault tolerant applications



SUPERVISORS

- Built on top of monitors
- Supervise actors
- Automatically kills/restarts actors based on rules
- Can form supervision tree



Supervisor Tree

APPLICATION

- A component which provides a service
- Started by runtime (VM)

RELEASES

- Unit for deployment
- Can have multiple applications
- Hot code swapping support

ELIXIR BASICS

- Basic Types
- Pattern Matching
- Modules and Functions
- Metaprogramming
- Polymorphism
- Error Handling

BASIC TYPES

```
iex> "hello"   # <- A String
"hello"

iex> :hello    # <- An atom(Constant String)
:hello

iex> count = 5 # <- An Integer
5

iex> count+1   # <- Add 1 to count
6

iex> map = %{"Martin" => "QA", "Niklas" => 1, 2 => "Dominik"} # <- A Map/dictionary
%{200 => "Dominik", "Martin" => "QA", "Niklas" => 100}

iex> map["Martin"] # <- Access value of Map
"QA"

iex> names = {"Jasbir", "Stanislav", "Ramesh", "Jelome"} # <- A Tuple
{"Jasbir", "Stanislav", "Ramesh", "Jelome"}

iex> elem(names, 1) # <- Access a tuple (ordered)
"Stanislav"
```

PATTERN MATCHING

= is the matching operator.

```
iex> {:longhair, name} = {:longhair, "Matthias"}
{:longhair, "Matthias"}  # <- Pattern match success

iex> name
"Matthias"

iex> {:longhair, name} = {:shorthair, "Tino"} # <- Pattern match fails
(MatchError) no match of right hand side value: {:shorthair, "Tino"}
```

Without pattern matching

```
  hair, name = ("longhair", "Matthias")
  if hair == "longhair":
    # Do something with *name*
```

MORE PATTERN MATCHING

```
iex> team = {"milti", "kaspar"}
{"milti", "kaspar"}

iex> case team do
 {"milti", x} -> x
 :johannes -> "Football"
end

"kaspar"
```

MODULES, FUNCTIONS AND LAMBDAS

```
defmodule Factorial do # <- Defines a module, with defmodule

 # Functions are defined with def
 def factorial(0), do: 1 # <- Pattern matching in arguments
 def factorial(n), do: n * factorial(n - 1) # <- Recursion
end
```

```
iex> Factorial.factorial(5)
120
```

```
iex> company = "Tb"
"Tb"

# Anonymous functions are defined with fn and forms a closure
iex> sayName = fn p -> IO.puts("I am #{p} in #{company}") end # <- Anonymous function
```

```
iex> sayName.("Marco") # <- Note '.' after sayName
I am Marco in Tb # <- 'Tb' came from closure
```

METAPROGRAMMING

- Build your own DSL
- Easily extend host language

```
defmodule More do
  defmacro notif(condition, do: body) do
    quote do
      if !unquote(condition), do: unquote(body)
    end
  end
end
```

```
iex(1)> if 5 < 6, do: IO.puts("hello world")
hello world

iex(2)> import More
iex(3)> notif 5 > 6, do: IO.puts("hello world") # <- Seamlessly extendable language
hello world
```

PROTOCOLS (INTERFACE)

- For polymorphism
- Can extend existing APIs for new datatypes.

```
defprotocol SayName do
  @doc "Says the type of data structure"
  def name(n)
end

defmodule User do
  defstruct [:name] # <- Define a struct (similar to Map)
end

# If you want to implement SayNAme protocol, implement name function

defimpl SayName, for: Map do # <- Implement SayName for Map datatype
  def name(_), do: "I am MAP"
end

defimpl SayName, for: User do
  def name(user) do
    "I am #{user.name}" # <- This is a User struct
  end
end
```

```
SayName.name(%{ok:  "A map"})
"I am MAP"

SayName.name(%User{name: "Sven"})
"I am Sven"
```

PROTOCOLS AS UNIFORM INTERFACES

Enum.map(data, f) - Applies function **f** over **data**

```
iex> names1 = %{"1" => "Whitrapee", "2" => "Goran", "3" => "Fabian"}
%{"1" => "Whitrapee", "2" => "Goran", "3" => "Fabian"} # <- It's a Map

iex> names2 = ["Jasbir", "Rustam", "Vlad"]
["Jasbir", "Rustam", "Vlad"] # <- It's a list

iex> Enum.map(names1, fn {k,v} -> "☺ #{v}" end)
["☺ Whitrapee", "☺ Goran", "☺ Fabian"]  # <- Enum.map works on Map datatype

iex> Enum.map(names2, fn n -> "☺ #{n}" end)
["☺ Jasbir", "☺ Rustam", "☺ Vlad"] # <- Enum.map works on List datatype
```

How does *Enum.map* work with both *Map* and *List* ?

Protocols

PROTOCOL BEHAVIOUR

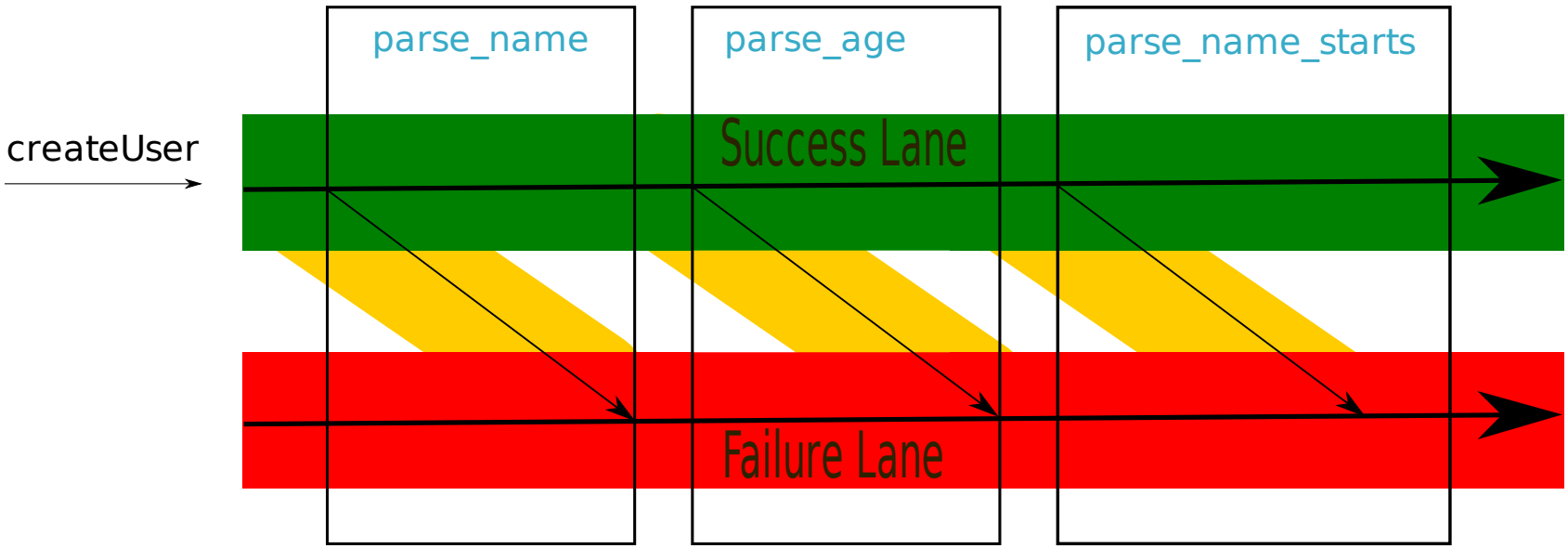
```
iex> i &Enum.map/2 # <- 'i' is a helper function for Inspection
  &Enum.map/2
Data type
  Function
Implemented protocols
  Enumerable, Inspect, IEx.Info

iex> i names1 # <- Inspect Map
Term
  %{"1" => "Whitrapee", "2" => "Goran", "3" => "Fabian"}
Data type
  Map
Implemented protocols
  Enumerable, Inspect, IEx.Info, Collectable

iex> i names2 # <- Inspect List
Term
  ["Jasbir", "Rustam", "Vlad"]
Data type
  List
Implemented protocols
  Enumerable, String.Chars, Inspect, IEx.Info, Collectable, List.Chars
```

Enum.map *internally calls* **Enumerable.reduce** *protocol, which if implemented makes the data type enumerable.*

ERROR HANDLING



- Railway-oriented programming
  - Two lanes - success lane and failure lane
  - Automatic switching of lanes

RAILWAY-ORIENTED PROGRAMMING

```elixir
defmodule Person do
  @moduledoc false
  defstruct name: nil, age: nil
  def create_user(params) do
    with {:ok, name} <- parse_name(params[:name]), # <- Follows a pipeline
         {:ok, age} <- parse_age(params[:age]),
         {:ok, name} <- parse_name_starts(params[:name]) do
      %Person{name: name, age: age}
    else
      error -> error # <- Executed when one of the above steps fail
    end
  end

  def parse_name(nil), do: {:error, "Name is required"}
  def parse_name(""), do: {:error, "Name is required"}
  def parse_name(name), do: {:ok, name}

  def parse_age(nil), do: {:error, "Age is required"}
  def parse_age(age) when age < 0, do: {:error, "Age cannot be negative"}
  def parse_age(age), do: {:ok, age}

  def parse_name_starts(name) do
    case String.starts_with?(name, "S") do
      true -> {:ok, name}
      false -> {:error, "Name should start with S"}
    end
  end
end
```

TRYING OUT

```
iex(1)> Person.create_user(%{name: "Sujay", age: 5})
%Person{age: 5, name: "Sujay"}
iex(2)> Person.create_user(%{name: "", age: 5})
{:error, "Name is required"}
iex(3)> Person.create_user(%{name: "Robert", age: -1})
{:error, "Age cannot be negative"}
iex(4)> Person.create_user(%{name: "sujay", age: 5})
{:error, "Name should start with S"}
```

COMMUNICATING ACTORS

- Create actors using **spawn**
- Send/Receive message using **send/receive**

```
defmodule PingPong do
  import :timer
  @time_interval 1000

  def ping_pong do
    receive do # <- Receive messages
      {:ping, sender, ping_id} -> # <- Pattern matching :D
        IO.puts("Ping received #{ping_id}")
        :timer.sleep(@timer_interval)
        send(sender, {:pong, self(), ping_id + 1})

      {:pong, sender, pong_id} ->
        IO.puts("Pong received #{pong_id}")
        :timer.sleep(1000)
        send(sender, {:ping, self(), pong_id + 1})
    after # <- Timeout functionality
      @timer_interval -> "No messages received"
    end

    ping_pong() # <- Recursive call
  end
end
```

```
iex(1)> one = spawn(PingPong, :ping_pong, []) # <- Create actor
iex(2)> two = spawn(PingPong, :ping_pong, [])
iex(3)> send(one, {:ping, two, 0}) # <- Start by sending message
Ping received 0
Pong received 1
```

DISTRIBUTED PROGRAMMING

- Connect and run computations on remote nodes
- Load code remotely

```
iex --sname foo@localhost # <- Start Node with name foo
```

```
iex --sname bar@localhost # <- Start Node with name bar
```

```
iex(bar@localhost)1> Node.connect :foo@localhost # <- Connect from bar to foo
true

iex(bar@localhost)2> Node.spawn(:"foo@localhost",
  fn -> IO.puts "Hello from #{Node.self()}" end) # <- Send code to execute on remote
#PID<10349.150.0>
Hello from foo@localhost
```

- Code can be loaded to all connected nodes using **nl(Module)**

```
iex(foo@localhost)> nl(Factorial)
{:ok,
 [
   {:foo@localhost, :loaded, Factorial},
 ]}
```

OPEN TELECOM PLATFORM (OTP)

OTP framework is a set of modules and standards designed to help build applications.

- Includes
  - GenServer
  - Supervisors
  - And more

GENSERVER (GENERIC SERVER)

- Holds state and exposes supported operations

```
defmodule Factorial do
  def factorial(0), do: 1
  def factorial(n), do: n * factorial(n - 1)
end
```

```
defmodule CoolerFactorialServer do
  use GenServer

  def init(_) do
    {:ok, %{count: 0}} # <- Initial Server State. Count of factorials computed.
  end

  # Starts a GenServer process running this module
  def start_link(worker_name) do
    GenServer.start_link(__MODULE__, [], name: String.to_atom("#{worker_name}"))
  end

  def handle_call({:factorial, number}, from, state) do # <- Handle :factorial msgs
    {:reply, {Factorial.factorial(number), state[:count]}, # <- Reply with factorial
      %{state | count: state[:count] + 1}} # <- and updates genserver state.
  end

  # Helper function for use by clients
  def get_factorial(pid, number) do
    GenServer.call(pid, {:factorial, number})
  end
end
```

SUPERVISING FACTORIAL SERVER

- Now we got a factorial server, but how do we handle crashing of these servers?. *Supervisors*

```
defmodule FactorialSupervisor do
  use Supervisor

  # Start a supervisor with current module
  def start_link(opts) do
    Supervisor.start_link(__MODULE__, :ok, opts)
  end

  # Initialization function, which generates list of children with names
  def init(:ok) do
    children =
      Enum.map(1..3, fn n ->
        Supervisor.child_spec(
          {CoolerFactorialServer, n},
          id: String.to_atom("#{n}")
        )
      end)

    # Starts supervisor, with one_for_one strategy.
    # This strategy restarts processes which crashed.
    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

WORKING OF SUPERVISOR

```
iex(1)> {:ok, pid} = FactorialSupervisor.start_link([])
{:ok, #PID<0.112.0>}

iex(2)> Supervisor.which_children(pid)
[
  {:"3", #PID<0.115.0>, :worker, [CoolerFactorialServer]},
  {:"2", #PID<0.114.0>, :worker, [CoolerFactorialServer]},
  {:"1", #PID<0.113.0>, :worker, [CoolerFactorialServer]}
]

iex(3)> GenServer.call(:"1", {:factorial, 5})
120

iex(4)> CoolerFactorialServer.get_factorial(
  Process.whereis(:"1"), "hello") # <- Call factorial with invalid data
** (exit) exited in: GenServer.call(#PID<0.113.0>, {:factorial, "hello"}, 5000)
    ** (EXIT) an exception was raised:
      ** (ArithmeticError) bad argument in arithmetic expression # <- Process crashed
          (f) lib/gensuper.ex:2: Factorial.factorial/1 ....

iex(5)> Supervisor.which_children(pid)
[
  {:"3", #PID<0.115.0>, :worker, [CoolerFactorialServer]},
  {:"2", #PID<0.114.0>, :worker, [CoolerFactorialServer]},
  {:"1", #PID<0.125.0>, :worker, [CoolerFactorialServer]} # <- Restarted with new PID
]
iex(6)> CoolerFactorialServer.get_factorial(:"1", 5)
{120, 0}

iex(7)> CoolerFactorialServer.get_factorial(:"1", 5)
{120, 1} # <- Counter incremented
```

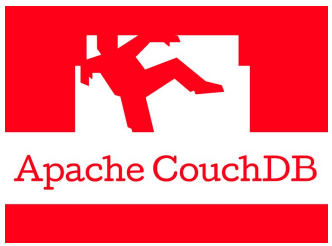
- Now we have a fault tolerant Factorial Server.

OBSERVABILITY

- Default Observer tool
  - Fine grained information about VM
  - Trace/kill/debug processes
  - And a lot more

```
iex> :observer.start()
```

PROJECTS USING BEAM/ELIXIR




MOZ


b/r
bleacher
report

OTHER GOODIES

- Mix/Hex/Exunit - Build/Package Manager/Testing
- Inbuilt formatter and rules
- Mature web development frameworks (Phoenix Framework)
- Embedded systems support (Nerves Project)
- Optional Typing (Dialyxir)
- Inbuilt cache/database (Mnesia, ETS)
- Checkout awesome elixir at github

SUMMARY

- Looked at motivation for Elixir
- Briefly introduced functional programming
- Took a look at elixir concepts and syntax
- Learned basics on building fault tolerant apps

REFERENCES

- https://twitter.com/reubenbond/status/662061791497744384?lang=en [1]
- https://www.techworld.com/apps-wearables/how-elixir-helped-bleacher-report-handle-8x-more-traffic-3653957/ [2]

THANK YOU

QUESTIONS?

PRACTICE TASK

UNIVERSAL SERVER

- A server which can be converted to a specific server
  - Can be converted to an
  - Factorial Server
  - *Enter you prefered Server here*
- Can be converted to specific with a *:become* message

A FACTORIAL SERVER

- A server which computes Factorial of a number

CONVERT UNIVERSAL TO FACTORIAL SERVER

- Step 1
  - Spawn a universal server
  - Send *become* Factorial
  - Send number and get back factorial
- Step 2
  - Spawn 10 Universal servers and convert to Factorial servers
  - Send number to all these servers and get results

GETTING STARTED

- Create new project
  - `bash mix new project_name`
- Run IEX with project code
  - `bash iex -S mix`

MAKE A NEW PROJECT

```
mix new server
```

A UNIVERSAL SERVER

```
defmodule UniversalServer do
 def listen() do
  receive do # <- Receive messages
   {:become, f} -> # <- Handle :become messages
    f.() # <- Runs received function
   _ -> "Fail in universal"
  end
 end
end
```

A FACTORIAL SERVER

```
defmodule FactorialServer do
 def factorial(0), do: 1
 def factorial(n), do: n * factorial(n - 1)

 def listen() do
  receive do # <- Listen for messages
   {caller, n} -> # <- Handle all messages as number
      send(caller, {:result, factorial(n)}) # <- Send result back
   x -> IO.puts(x)
  end

  listen()
 end
end
```

CONVERT TO FACTORIAL SERVER

```
iex --sname foo@localhost # <- Start Node foo
```

```
iex --sname bar@localhost -S mix # <- Start Node bar with mix
```

```
Node.connect :foo@localhost # <- Connect from bar to foo
iex(bar@localhost)1> Node.connect :foo@localhost
true

iex(bar@localhost)2> Node.list # <- List connected nodes
[:foo@localhost]

iex(bar@localhost)3> nl(UniversalServer) # <- Remote code loading
{:ok, [{:foo@localhost, :loaded, UniversalServer}]}

iex(bar@localhost)4> nl(FactorialServer) # <- Remote code loading
{:ok, [{:foo@localhost, :loaded, FactorialServer}]}

iex(bar@localhost)5> k = Node.spawn(:"foo@localhost",
  fn ()-> UniversalServer.listen() end) # <- Spawn universal server in remote
#PID<14566.98.0>

iex(bar@localhost)6> send k, {:become, fn ->
  FactorialServer.listen() end} # <- Send :become message
{:become, #Function<20.99386804/0 in :erl_eval.expr/5>}

iex(bar@localhost)7> send k, {self(), 5} # <- Send for factorial
{:factorial, #PID<0.116.0>, 5}

iex(bar@localhost)8> flush
{:result, 120} # <- Received result
:ok
```

MORE UNIVERSAL SERVERS (STEP 2)

```
iex> servers = Enum.map(1..10,
  fn x -> spawn(UniversalServer, :listen, []) end)

iex> Enum.map(servers, fn x -> send(x, {:become,
  fn -> FactorialServer.listen() end}) end)

iex> Enum.map(servers, fn x -> send(x, {self(), 10}) end)

iex> flush
```