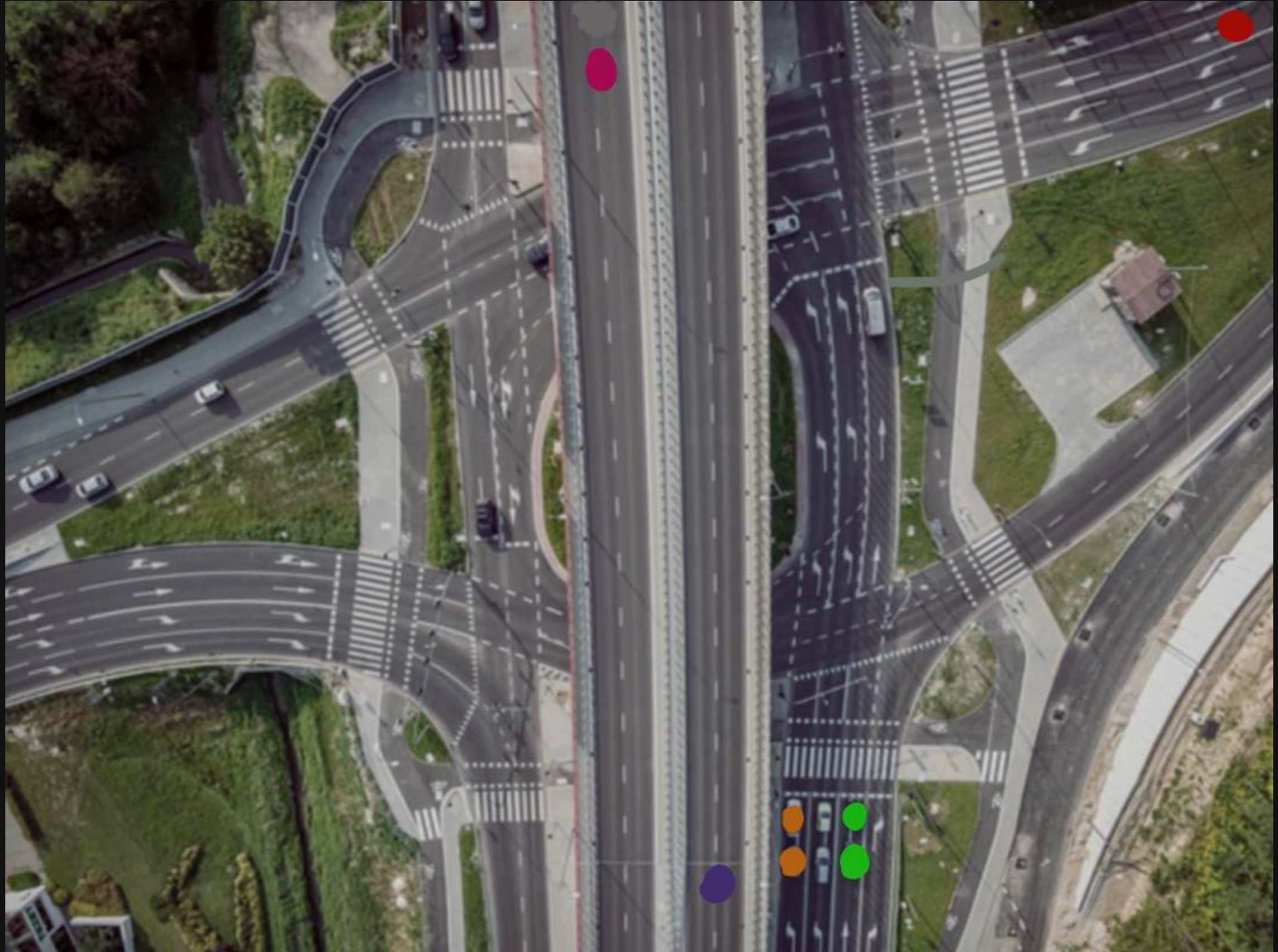
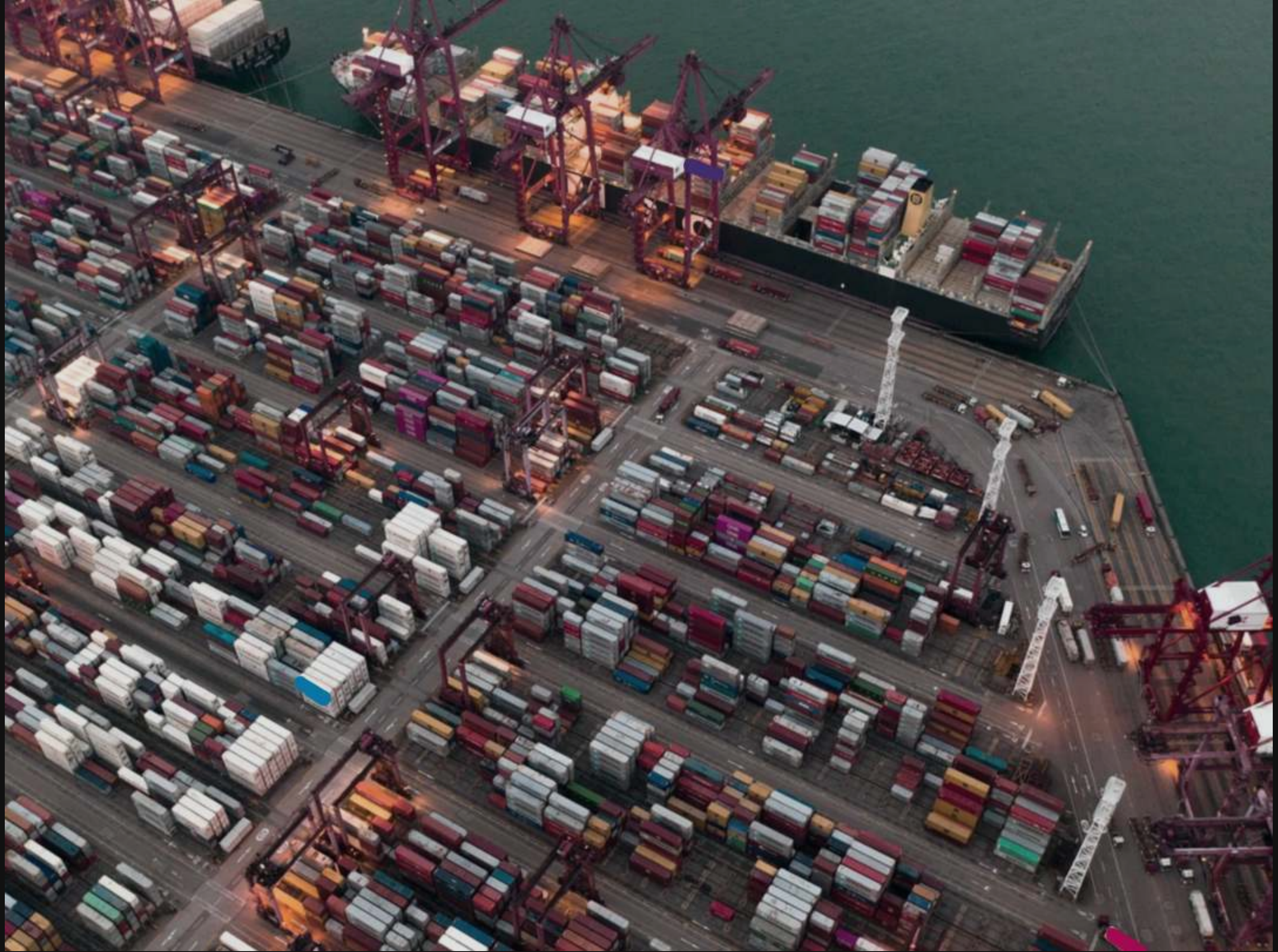
Is your world concurrent ?

How is your world speaking ?






LA MARCA SEGURA
Comunia Monterrey
adidas

Is your code concurrent ?

Telecoms systems

Making reliable distributed systems in the presence of software errors (Joe Armstrong, 2003)

1995 → First AXD301 released with Erlang

1998 → Erlang, Virtual Machine (BEAM) & Open Telecom Platform (OTP) Open Source

Creation of the Elixir language (José Valim, 2011)

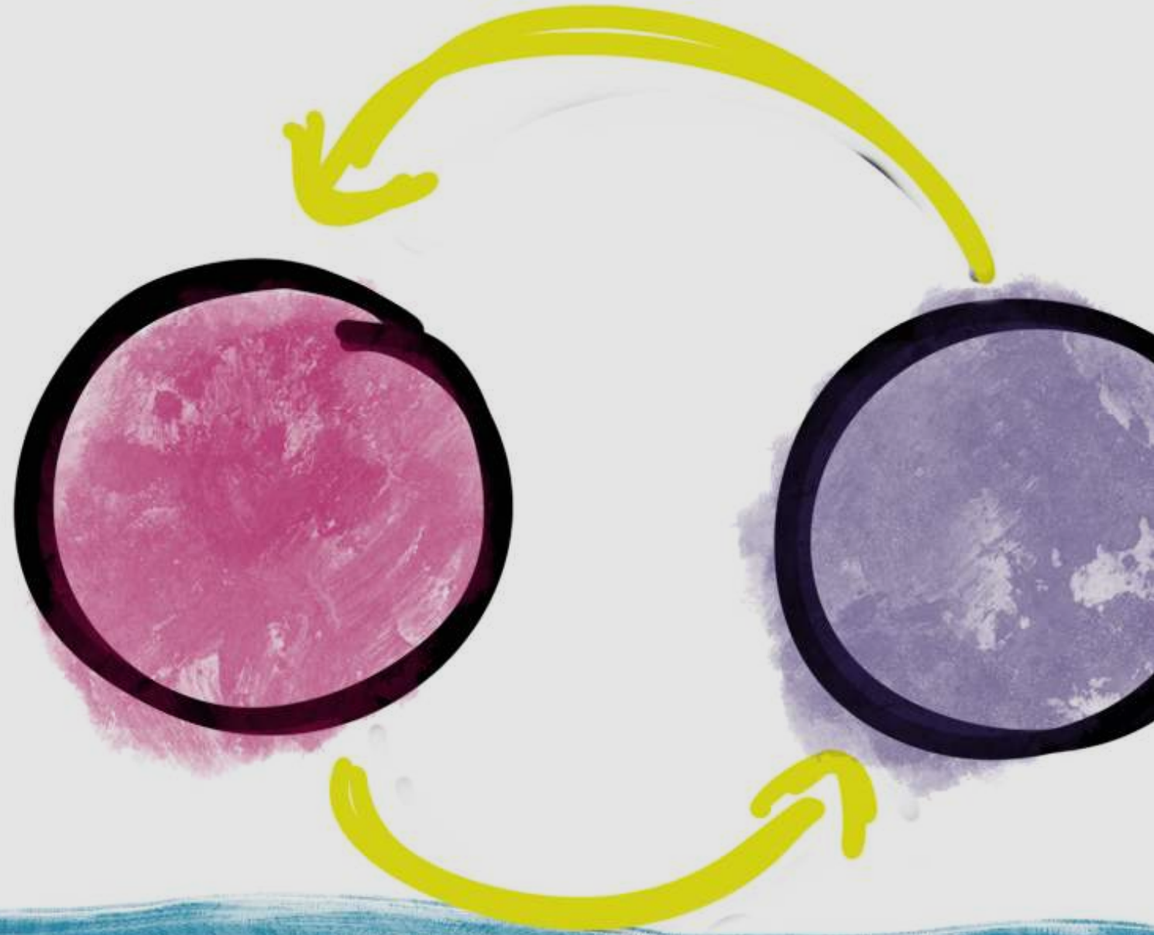
2014 → First production version running on the BEAM

2016 → Phoenix Framework first release

Fault tolerance
High availability
Distribution
Scalability

Concurrency

Shared Memory



Message passin

Concurrent message-passing ?

Concurrent message-passing ?

1. Language process = virtual machines

Concurrent message-passing ?

1. Language process = virtual machines
2. Strong process isolation

Concurrent message-passing ?

1. Language process = virtual machines
2. Strong process isolation
3. Each process has its own identity

Concurrent message-passing ?

1. Language process = virtual machines
2. Strong process isolation
3. Each process has its own identity
4. No shared state between processes

Concurrent message-passing ?

1. Language process = virtual machines
2. Strong process isolation
3. Each process has its own identity
4. No shared state between processes
5. Message passing with no guarantee of delivery

Concurrent message-passing ?

1. Language process = virtual machines
2. Strong process isolation
3. Each process has its own identity
4. No shared state between processes
5. Message passing with no guarantee of delivery
6. Processes can detect other processes failures

An architecture is “a way of thinking about the world.”,

An architecture is “a way of thinking about the world.”,

and the Actor Model is an intuitive architecture!

Maxime Janvier
SNOWCAMP 2023

ELIXIR

une potion pour construire vos microservices dans un monolithe

LANCEY
energy storage

AUTOCONSOMMATION / STOCKAGE D'ÉNERGIE /
CHAUFFAGE CONNECTÉ / PHOTOVOLTAÏQUE

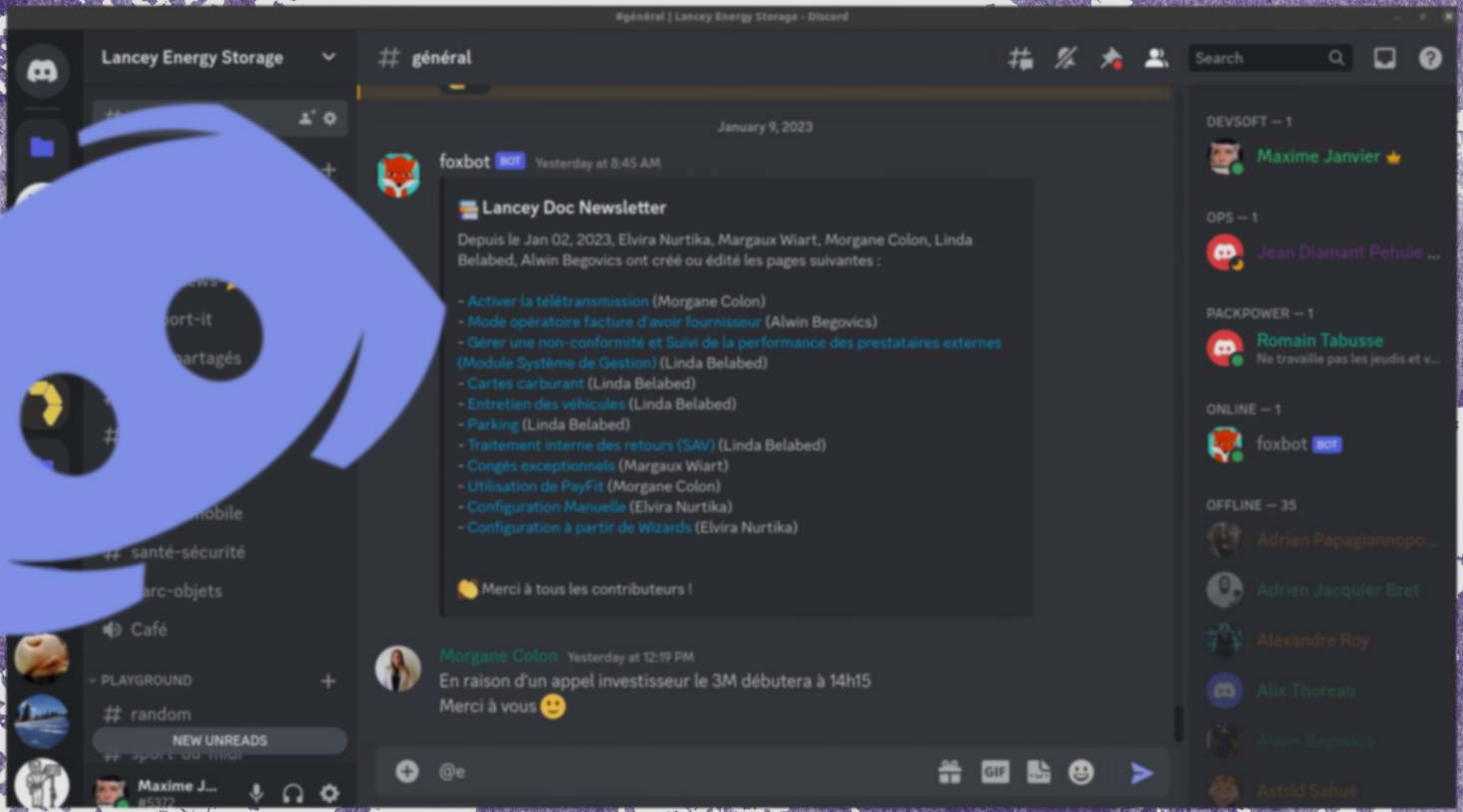
Une batterie de stockage intégrée dans un radiateur électrique intelligent : ça change tout !

Transition énergétique : LANCEY Energy Storage vous propose le seul système qui optimise l'autoconsommation photovoltaïque pour réduire la facture d'électricité. Améliorez le confort thermique grâce à un système de gestion totalement innovant, basé sur un radiateur électrique intelligent et avec batterie intégrée.


riak


RabbitMQ


WhatsApp
CHATS 3
Calls
Status
Ayesha Pawar
Best one there is!
14:24
Coffee & Consoles
Dario: Who's up for playi ...
13:37
Jihoon Seo
Hey, tried to call...
11:26
Shannon Brown
GIF
9:28
Baking Club
Yesterday
These sourdough starters are awf...
Myka Mercado
Yesterday
Let's chat about it another time.
Your personal messages are end-to-end encrypted
Select contact
256 contacts
Create
New group
New contact
New community
Discover
Businesses
Contacts on WhatsApp
Aby John
Hey there! I'm using WhatsApp.
Adam Benton
Adam Cain
Available
Adele Diver
Hey there! I'm using WhatsApp.
Aditya Kulkarni
Hey there! I'm using WhatsApp.
Aisha
Deep down, I'm really shallow
Businesses
Search all businesses
Nearby
Vila Madalena
Restaurant
Cafeteria
Grocery store
Apparel & clothing
Verified accounts
Markt
Mangata & Gallo
Jasper's Market
Samantha Weber
Location: Vila Madalena
Grocery store
Near Vila Madalena
Categories
Open now
Has catalog
Irmãos Silva
Mercearia
Open until 22:00
100 m • Rua João Santucci, 270 - Vi...
Multi Barato
Mercearia
Open until 22:00
200 m • Av. Presidente Wilson, 1120...
Viet Market
Mercearia
Open until 22:00
220 m • Rodovia Raposo Tavares...


#général | Lancey Energy Storage - Discord
Lancey Energy Storage
général
Search
January 9, 2023
foxbot BOT Yesterday at 8:45 AM
Lancey Doc Newsletter
Depuis le Jan 02, 2023, Elvira Nurtika, Margaux Wiart, Morgane Colon, Linda Belabed, Alwin Begovics ont créé ou édité les pages suivantes :
- Activer la télétransmission (Morgane Colon)
- Mode opératoire facture d'avoir fournisseur (Alwin Begovics)
- Gérer une non-conformité et Suivi de la performance des prestataires externes (Module Système de Gestion) (Linda Belabed)
- Cartes carburant (Linda Belabed)
- Entretien des véhicules (Linda Belabed)
- Parking (Linda Belabed)
- Traitement interne des retours (SAV) (Linda Belabed)
- Congés exceptionnels (Margaux Wiart)
- Utilisation de PayFit (Morgane Colon)
- Configuration Manuelle (Elvira Nurtika)
- Configuration à partir de Wizards (Elvira Nurtika)
Merci à tous les contributeurs !
Morgane Colon Yesterday at 12:19 PM
En raison d'un appel investisseur le 3M débutera à 14h15
Merci à vous
@e
santé-sécurité
Café
PLAYGROUND
random
NEW UNREADS
Maxime J...
DEVSOFT — 1
Maxime Janvier
OPS — 1
PACKPOWER — 1
Romain Tabusse
ONLINE — 1
foxbot BOT
OFFLINE — 35
Adrien Jacquier Bret
Alexandre Roy
Alix Thereau
Astrid Sahut

Your core is functional,

Your core is functional,
your world is concurrent,

Your core is functional,
your world is concurrent,
let's build an app !

Your core is functional

Dynamic language

Dynamic language
Data is immutable

Dynamic language

Data is immutable

Functions are pure

```elixir
1  1                    # Integers
2  1.4                  # Floats
3  nil                  # Null value
4  true                 # Booleans
5  false                #
6  :symbol              # Atoms (💜)
7  {1, 2, 3}            # Tuples
8  [1, 2, 3]            # Lists
9  %{a: 1, b: 2, c: 3}  # Maps
10 "string"             # Strings
```

```elixir
1  1                    # Integers
2  1.4                  # Floats
3  nil                  # Null value
4  true                 # Booleans
5  false                #
6  :symbol              # Atoms ( )
7  {1, 2, 3}            # Tuples
8  [1, 2, 3]            # Lists
9  %{a: 1, b: 2, c: 3}  # Maps
10 "string"             # Strings
```

```elixir
1  1                    # Integers
2  1.4                  # Floats
3  nil                  # Null value
4  true                 # Booleans
5  false                #
6  :symbol              # Atoms ( )
7  {1, 2, 3}            # Tuples
8  [1, 2, 3]            # Lists
9  %{a: 1, b: 2, c: 3}  # Maps
10 "string"             # Strings
```

```elixir
1  1                    # Integers
2  1.4                  # Floats
3  nil                  # Null value
4  true                 # Booleans
5  false                #
6  :symbol              # Atoms (💜)
7  {1, 2, 3}            # Tuples
8  [1, 2, 3]            # Lists
9  %{a: 1, b: 2, c: 3} # Maps
10 "string"             # Strings
```

```elixir
1  1                    # Integers
2  1.4                  # Floats
3  nil                  # Null value
4  true                 # Booleans
5  false                #
6  :symbol              # Atoms ( )
7  {1, 2, 3}            # Tuples
8  [1, 2, 3]            # Lists
9  %{a: 1, b: 2, c: 3}  # Maps
10 "string"             # Strings
```

```elixir
1  1                    # Integers
2  1.4                  # Floats
3  nil                  # Null value
4  true                 # Booleans
5  false                #
6  :symbol              # Atoms ( )
7  {1, 2, 3}            # Tuples
8  [1, 2, 3]            # Lists
9  %{a: 1, b: 2, c: 3}  # Maps
10 "string"             # Strings
```

```elixir
1  1                    # Integers
2  1.4                  # Floats
3  nil                  # Null value
4  true                 # Booleans
5  false                #
6  :symbol              # Atoms ( )
7  {1, 2, 3}            # Tuples
8  [1, 2, 3]            # Lists
9  %{a: 1, b: 2, c: 3} # Maps
10 "string"             # Strings
```

```elixir
1  1                    # Integers
2  1.4                  # Floats
3  nil                  # Null value
4  true                 # Booleans
5  false                #
6  :symbol              # Atoms ( )
7  {1, 2, 3}            # Tuples
8  [1, 2, 3]            # Lists
9  %{a: 1, b: 2, c: 3}  # Maps
10 "string"             # Strings
```

```elixir
1  1                    # Integers
2  1.4                  # Floats
3  nil                  # Null value
4  true                 # Booleans
5  false                #
6  :symbol              # Atoms (💜)
7  {1, 2, 3}            # Tuples
8  [1, 2, 3]            # Lists
9  %{a: 1, b: 2, c: 3}  # Maps
10 "string"             # Strings
```

Modules

Modules

```
defmodule MyModule do
end
```

Structures

Structures

```
defmodule Item do
    defstruct title: "", price: 0.0
end
```

Structures

```
defmodule Item do
    defstruct title: "", price: 0.0
end
```

```
iex> %Item{items: "apple", price: 1.4}
```

Functions

Functions

```elixir
defmodule MyModule do
    def say_hello do
        IO.puts "Hello!"
    end
end
```

Functions

```
1 defmodule MyModule do
2     def say_hello do
3         IO.puts "Hello!"
4     end
5 end
```

Functions

```elixir
defmodule MyModule do
    def say_hello do
        IO.puts "Hello!"
    end
end
```

```
iex> MyModule.say_hello
Hello!
```

Functions

Functions

```
1 defmodule Rectangle do
2     def area(a, b) do
3         a * b
4     end
5
6     def area(a) do
7         area(a, a)
8     end
9 end
```

Functions

```
1 defmodule Rectangle do
2     def area(a, b) do
3         a * b
4     end
5
6     def area(a) do
7         area(a, a)
8     end
9 end
```

Functions

```elixir
defmodule Rectangle do
    def area(a, b) do
        a * b
    end

    def area(a) do
        area(a, a)
    end
end
```

Functions

```elixir
defmodule Rectangle do
    def area(a, b) do
        a * b
    end

    def area(a) do
        area(a, a)
    end
end
```

Functions

```elixir
defmodule Rectangle do
    def area(a, b) do
        a * b
    end

    def area(a) do
        area(a, a)
    end
end
```

```
iex> Rectangle.area(3, 4)
12

iex> Rectangle.area(5)
25
```

Pipelines

```
f(g(h(•, •), •), •)
```

```
• |> h(•) |> g(•) |> f(•)
```

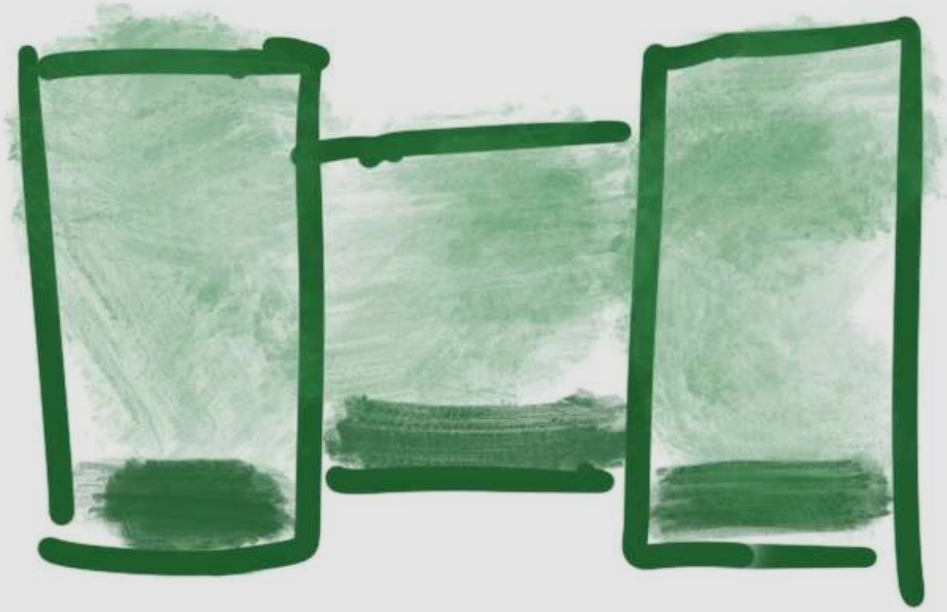
```elixir
defmodule Maths do
    def oneliner(n) do
        Enum.sum( Enum.map(Enum.filter(1..n, fn x -> Integer.is_odd(x) end),
        fn x -> x * x end))
    end

    def pipeline(n) do
        1..n
        |> Enum.filter(fn x -> Integer.is_odd(x) end)
        |> Enum.map(fn x -> x * x end)
        |> Enum.sum()
    end
end
```

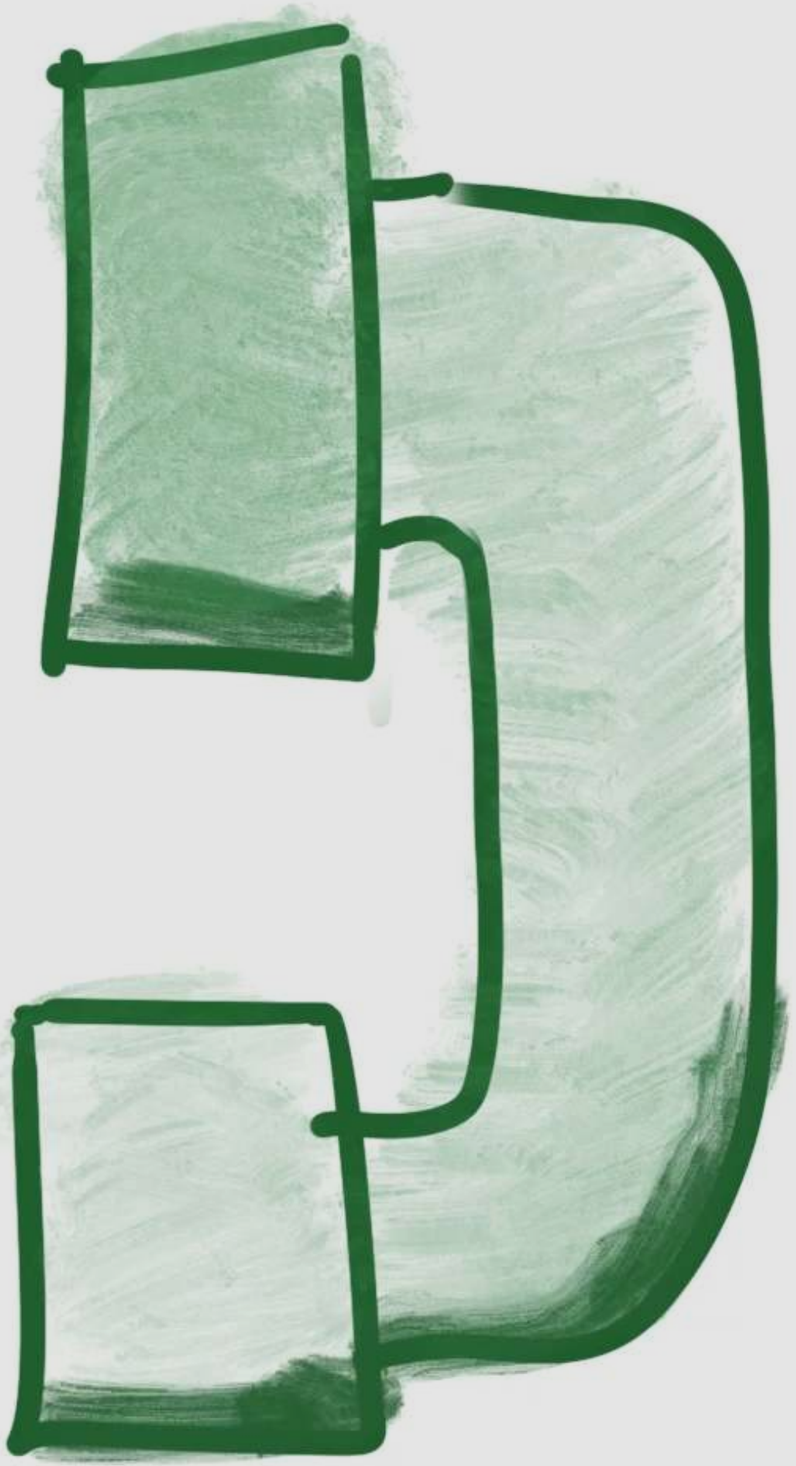
```elixir
defmodule Maths do
    def oneliner(n) do
        Enum.sum( Enum.map(Enum.filter(1..n, fn x -> Integer.is_odd(x) end),
        fn x -> x * x end))
    end

    def pipeline(n) do
        1..n
        |> Enum.filter(fn x -> Integer.is_odd(x) end)
        |> Enum.map(fn x -> x * x end)
        |> Enum.sum()
    end
end
```

```elixir
defmodule Maths do
    def oneliner(n) do
        Enum.sum( Enum.map(Enum.filter(1..n, fn x -> Integer.is_odd(x) end),
        fn x -> x * x end))
    end

    def pipeline(n) do
        1..n
        |> Enum.filter(fn x -> Integer.is_odd(x) end)
        |> Enum.map(fn x -> x * x end)
        |> Enum.sum()
    end
end
```

```elixir
defmodule Maths do
    def oneliner(n) do
        Enum.sum( Enum.map(Enum.filter(1..n, fn x -> Integer.is_odd(x) end),
        fn x -> x * x end))
    end

    def pipeline(n) do
        1..n
        |> Enum.filter(fn x -> Integer.is_odd(x) end)
        |> Enum.map(fn x -> x * x end)
        |> Enum.sum()
    end
end
```

```
iex> Maths.oneliner(100)
338350

iex> Maths.pipeline(100)
338350
```





Pattern matching

```
iex> {:ok, b, c} = {:ok, 2, 3}
{:ok, 2, 3}
iex> b
2
iex> c
3
```

```
iex> {:ok, b, c} = {:ok, 2, 3}
{:ok, 2, 3}
iex> b
2
iex> c
3
```

```
iex> {:ok, b, c} = {:nope, 2, 3}
** (MatchError) no match of right hand side value: {:nope, 2, 3}
```

```
iex> [head | tail] = [1, 2, 3]
[1, 2, 3]
iex> head
1
iex> tail
[2, 3]
```

```
iex> [head | tail] = [1, 2, 3]
[1, 2, 3]
iex> head
1
iex> tail
[2, 3]
```

```
iex> item = %Item{price: 1.3}
iex> %Item{price: p} = item
iex> p
1.3
```

```elixir
defmodule Sums do
    def sum([]), do: 0.0
    def sum([%Item{price: price} | items]), do: price + sum(items)

    def sum([]), do: 0.0
    def sum([head | tail]), do: head + sum(tail)

    def sum(_), do: -1.0
end
```

```elixir
defmodule Sums do
    def sum([]), do: 0.0
    def sum([%Item{price: price} | items]), do: price + sum(items)

    def sum([]), do: 0.0
    def sum([head | tail]), do: head + sum(tail)

    def sum(_), do: -1.0
end
```

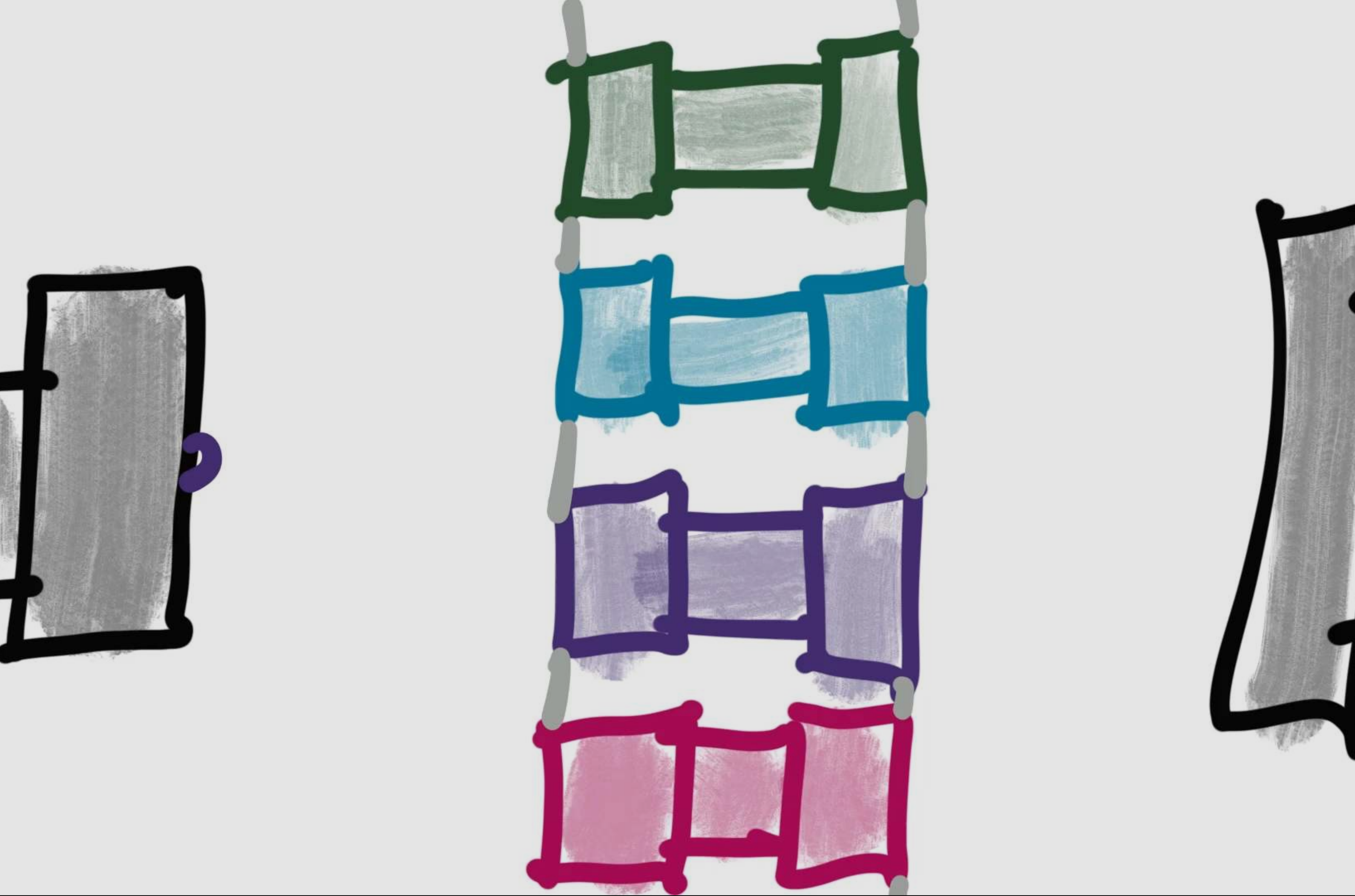
```elixir
defmodule Sums do
    def sum([]), do: 0.0
    def sum([%Item{price: price} | items]), do: price + sum(items)

    def sum([]), do: 0.0
    def sum([head | tail]), do: head + sum(tail)

    def sum(_), do: -1.0
end
```

```elixir
defmodule Sums do
    def sum([]), do: 0.0
    def sum([%Item{price: price} | items]), do: price + sum(items)

    def sum([]), do: 0.0
    def sum([head | tail]), do: head + sum(tail)

    def sum(_), do: -1.0
end
```

```elixir
defmodule Sums do
    def sum([]), do: 0.0
    def sum([%Item{price: price} | items]), do: price + sum(items)

    def sum([]), do: 0.0
    def sum([head | tail]), do: head + sum(tail)

    def sum(_), do: -1.0
end
```

```elixir
defmodule Sums do
    def sum([]), do: 0.0
    def sum([%Item{price: price} | items]), do: price + sum(items)

    def sum([]), do: 0.0
    def sum([head | tail]), do: head + sum(tail)

    def sum(_), do: -1.0
end
```

```
iex> Sums.sum([%Item{price: 2.1}, %Item{price: 3.4}, %Item{price: 1.0}])
6.5

iex> Sums.sum([1.0, 2.0, 3.0])
6.0

iex> Sums.sum(%{a: 1})
-1.0
```



Elixir is functional and dynamic

Elixir is functional and dynamic
Data is immutable

Elixir is functional and dynamic

Data is immutable

Pipelines + pattern matching = 💜

Your world is concurrent

Data is immutable

Data is immutable

Everything run in a process

Data is immutable

Everything run in a process

Running a process is cheap

Data is immutable

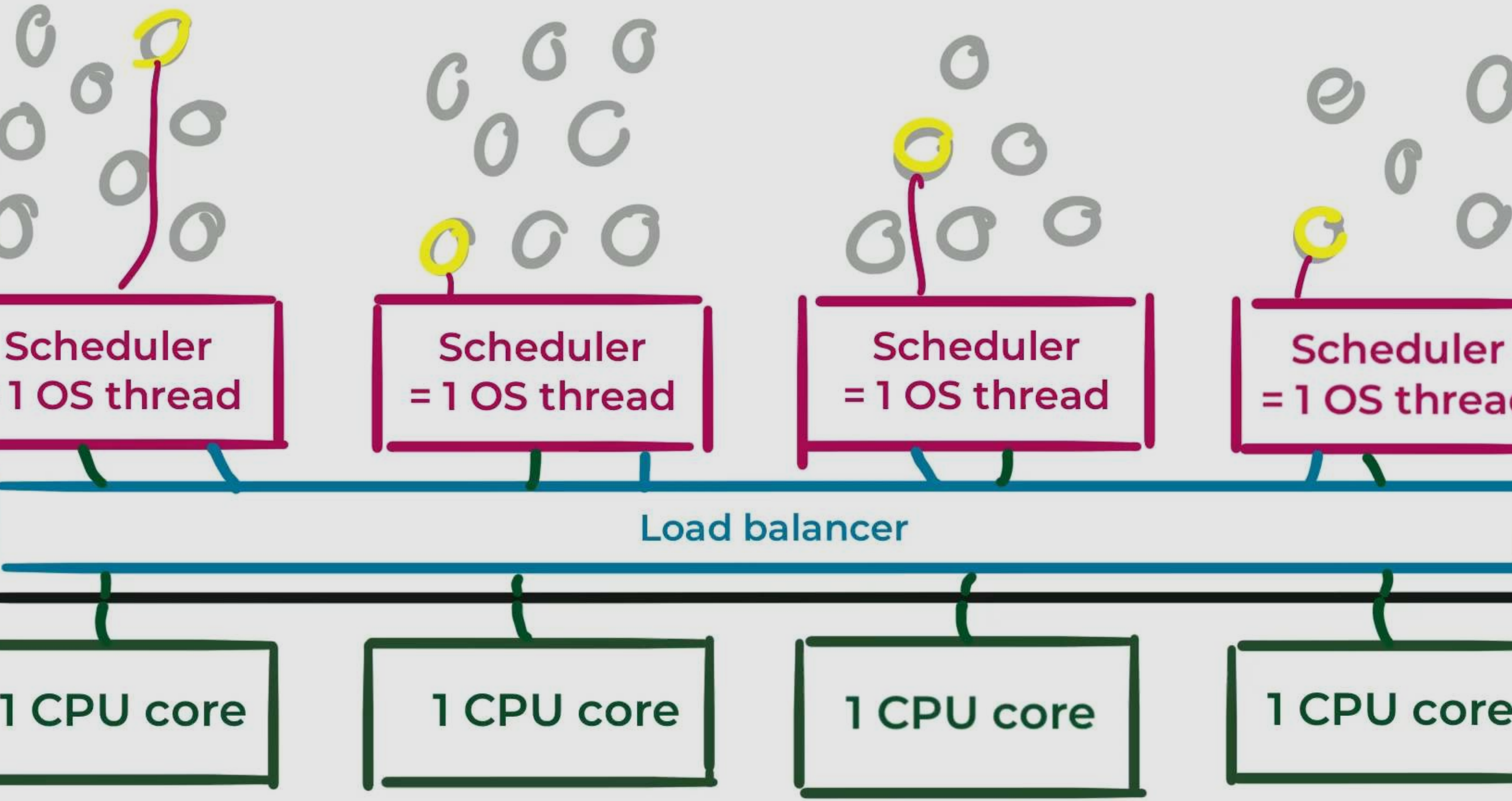
Everything run in a process

Running a process is cheap

Asynchronous messages everywhere

How is the BEAM working ?

BEAM = 1 OS proce



What is a process ?

Start one process → spawn

Start one process → spawn

```
iex> pid = spawn(fn -> IO.puts("Hi, I am a cell!") end)
Hi, I am a cell!
#PID<0.141.0>
```

Start one process → spawn

```
iex> pid = spawn(fn -> IO.puts("Hi, I am a cell!") end)
Hi, I am a cell!
#PID<0.141.0>
```

```
iex> Process.alive?(pid)
false
```

Start an infinite process → receive

Start an infinite process → receive

```elixir
defmodule Cell do
    def loop(count) do
        receive do
            :stop ->
                exit(:normal)
            message ->
                IO.puts("#{n}: #{message}")
                loop(count - 1)
        end
    end
end
```

Start an infinite process → receive

```elixir
defmodule Cell do
    def loop(count) do
        receive do
            :stop ->
                exit(:normal)
            message ->
                IO.puts("#{n}: #{message}")
                loop(count - 1)
        end
    end
end
```

Start an infinite process → receive

```elixir
defmodule Cell do
    def loop(count) do
        receive do
            :stop ->
                exit(:normal)
            message ->
                IO.puts("#{n}: #{message}")
                loop(count - 1)
        end
    end
end
```

Start an infinite process → receive

```elixir
defmodule Cell do
    def loop(count) do
        receive do
            :stop ->
                exit(:normal)
            message ->
                IO.puts("#{n}: #{message}")
                loop(count - 1)
        end
    end
end
```

Start an infinite process → receive

```elixir
defmodule Cell do
    def loop(count) do
        receive do
            :stop ->
                exit(:normal)
            message ->
                IO.puts("#{n}: #{message}")
                loop(count - 1)
        end
    end
end
```

Start an infinite process → receive

```elixir
defmodule Cell do
    def loop(count) do
        receive do
            :stop ->
                exit(:normal)
            message ->
                IO.puts("#{n}: #{message}")
                loop(count - 1)
        end
    end
end
```

Start an infinite process → receive

```elixir
defmodule Cell do
    def loop(count) do
        receive do
            :stop ->
                exit(:normal)
            message ->
                IO.puts("#{n}: #{message}")
                loop(count - 1)
        end
    end
end
```

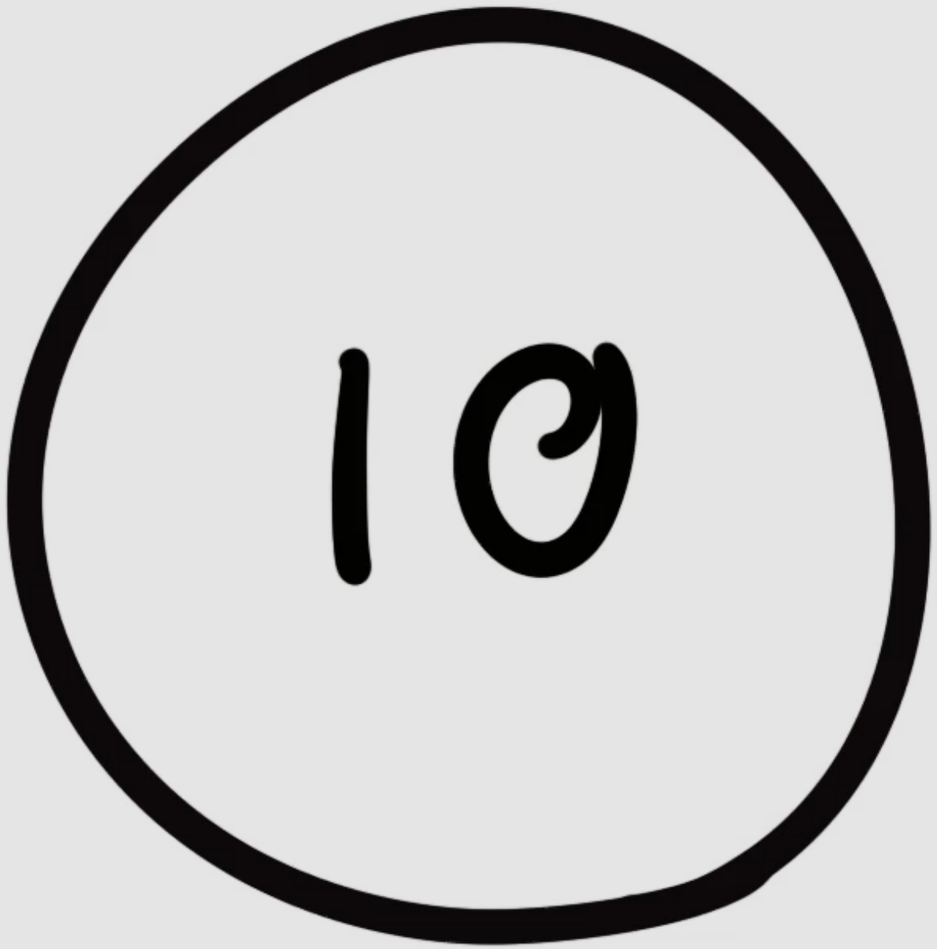
Talk to a process → send

Start an infinite process → receive

```elixir
defmodule Cell do
    def loop(count) do
        receive do
            :stop ->
                exit(:normal)
            message ->
                IO.puts("#{n}: #{message}")
                loop(count - 1)
        end
    end
end
```

Talk to a process → send

```elixir
pid = spawn(fn -> loop(10) end)
send(pid, "Hello")
send(pid, :stop)
```


10

Start a thousand processes !

Start a thousand processes !

```elixir
defmodule Clock do
    def start(count) do
        pid = spawn(fn -> loop(name, count) end)
        Process.send_after(pid, :tick, 1_000)
    end

    def loop(_, 0), do: IO.puts("Stop")
    def loop(name, count) do
        receive do
            :tick ->
                IO.puts("#{name}: Tick #{count}")
                Process.send_after(self(), :tick, 500 + :rand.uniform(100))
                loop(name, count - 1)
        end
    end
end
```

Start a thousand processes !

```elixir
defmodule Clock do
    def start(count) do
        pid = spawn(fn -> loop(name, count) end)
        Process.send_after(pid, :tick, 1_000)
    end

    def loop(_, 0), do: IO.puts("Stop")
    def loop(name, count) do
        receive do
            :tick ->
                IO.puts("#{name}: Tick #{count}")
                Process.send_after(self(), :tick, 500 + :rand.uniform(100))
                loop(name, count - 1)
        end
    end
end
```

Start a thousand processes !

```elixir
defmodule Clock do
    def start(count) do
        pid = spawn(fn -> loop(name, count) end)
        Process.send_after(pid, :tick, 1_000)
    end

    def loop(_, 0), do: IO.puts("Stop")
    def loop(name, count) do
        receive do
            :tick ->
                IO.puts("#{name}: Tick #{count}")
                Process.send_after(self(), :tick, 500 + :rand.uniform(100))
                loop(name, count - 1)
        end
    end
end
```

Start a thousand processes !

```elixir
defmodule Clock do
    def start(count) do
        pid = spawn(fn -> loop(name, count) end)
        Process.send_after(pid, :tick, 1_000)
    end

    def loop(_, 0), do: IO.puts("Stop")
    def loop(name, count) do
        receive do
            :tick ->
                IO.puts("#{name}: Tick #{count}")
                Process.send_after(self(), :tick, 500 + :rand.uniform(100))
                loop(name, count - 1)
        end
    end
end
```

```elixir
for i <- 1..1000 do
    Clock.start("clock-#{i}", 10)
end
```

💜

Genserver

💜

Genserver

handle_call(message, state) -> state'

💜

Genserver

handle_call(message, state) -> state'

handle_cast(message, from, state) -> state'

```elixir
defmodule Cell do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

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

    def handle_call(:stop, _from, count) do
        {:stop, :normal, count}
    end

    def handle_cast(message, count) do
        IO.puts("#{count}: #{message}")
        {:noreply, count - 1}
    end
end
```

```elixir
defmodule Cell do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

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

    def handle_call(:stop, _from, count) do
        {:stop, :normal, count}
    end

    def handle_cast(message, count) do
        IO.puts("#{count}: #{message}")
        {:noreply, count - 1}
    end
end
```

```elixir
defmodule Cell do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

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

    def handle_call(:stop, _from, count) do
        {:stop, :normal, count}
    end

    def handle_cast(message, count) do
        IO.puts("#{count}: #{message}")
        {:noreply, count - 1}
    end
end
```

```elixir
defmodule Cell do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

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

    def handle_call(:stop, _from, count) do
        {:stop, :normal, count}
    end

    def handle_cast(message, count) do
        IO.puts("#{count}: #{message}")
        {:noreply, count - 1}
    end
end
```

```elixir
defmodule Cell do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

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

    def handle_call(:stop, _from, count) do
        {:stop, :normal, count}
    end

    def handle_cast(message, count) do
        IO.puts("#{count}: #{message}")
        {:noreply, count - 1}
    end
end
```

```elixir
defmodule Clock do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

    def init(count) do
        Process.send_after(self(), :tick, 1000)
        {:ok, count}
    end

    def handle_info(:tick, 0) do
        IO.puts("Stop")
        {:stop, :normal, 0}
    end

    def handle_info(:tick, count) do
        IO.puts("Tick #{count}")
        Process.send_after(self(), :tick, 500 + :rand.uniform(100))
        {:noreply, count - 1}
    end
end
```

```elixir
defmodule Clock do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

    def init(count) do
        Process.send_after(self(), :tick, 1000)
        {:ok, count}
    end

    def handle_info(:tick, 0) do
        IO.puts("Stop")
        {:stop, :normal, 0}
    end

    def handle_info(:tick, count) do
        IO.puts("Tick #{count}")
        Process.send_after(self(), :tick, 500 + :rand.uniform(100))
        {:noreply, count - 1}
    end
end
```

```elixir
defmodule Clock do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

    def init(count) do
        Process.send_after(self(), :tick, 1000)
        {:ok, count}
    end

    def handle_info(:tick, 0) do
        IO.puts("Stop")
        {:stop, :normal, 0}
    end

    def handle_info(:tick, count) do
        IO.puts("Tick #{count}")
        Process.send_after(self(), :tick, 500 + :rand.uniform(100))
        {:noreply, count - 1}
    end
end
```

```elixir
defmodule Clock do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

    def init(count) do
        Process.send_after(self(), :tick, 1000)
        {:ok, count}
    end

    def handle_info(:tick, 0) do
        IO.puts("Stop")
        {:stop, :normal, 0}
    end

    def handle_info(:tick, count) do
        IO.puts("Tick #{count}")
        Process.send_after(self(), :tick, 500 + :rand.uniform(100))
        {:noreply, count - 1}
    end
end
```

```elixir
defmodule Clock do
    use GenServer

    def start_link(count) do
        GenServer.start_link(__MODULE__, count)
    end

    def init(count) do
        Process.send_after(self(), :tick, 1000)
        {:ok, count}
    end

    def handle_info(:tick, 0) do
        IO.puts("Stop")
        {:stop, :normal, 0}
    end

    def handle_info(:tick, count) do
        IO.puts("Tick #{count}")
        Process.send_after(self(), :tick, 500 + :rand.uniform(100))
        {:noreply, count - 1}
    end
end
```

Link two processes

Link two processes

```elixir
current = self()

spawn_link(fn ->
  for i <- 1..5 do
    send(current, i)
  end
  exit(:normal)
end)

for _ <- 1..5 do
  receive do
    msg -> IO.puts(msg)
  end
end
```

Link two processes

```elixir
current = self()

spawn_link(fn ->
  for i <- 1..5 do
    send(current, i)
  end
  exit(:normal)
end)

for _ <- 1..5 do
  receive do
    msg -> IO.puts(msg)
  end
end
```

Link two processes

```elixir
current = self()

spawn_link(fn ->
  for i <- 1..5 do
    send(current, i)
  end
  exit(:normal)
end)

for _ <- 1..5 do
  receive do
    msg -> IO.puts(msg)
  end
end
```

Link two processes

```elixir
current = self()

spawn_link(fn ->
  for i <- 1..5 do
    send(current, i)
  end
  exit(:normal)
end)

for _ <- 1..5 do
  receive do
    msg -> IO.puts(msg)
  end
end
```

Link two processes

```elixir
current = self()

spawn_link(fn ->
  for i <- 1..5 do
    send(current, i)
  end
  exit(:normal)
end)

for _ <- 1..5 do
  receive do
    msg -> IO.puts(msg)
  end
end
```

```
1
2
3
4
5
```

```elixir
current = self()

spawn_link(fn ->
  for i <- 1..5 do
    send(current, i)
  end
  exit(:noooooo)
end)

for _ <- 1..5 do
  receive do
    msg -> IO.puts(msg)
  end
end
```

```elixir
current = self()

spawn_link(fn ->
  for i <- 1..5 do
    send(current, i)
  end
  exit(:nooooooo)
end)

for _ <- 1..5 do
  receive do
    msg -> IO.puts(msg)
  end
end
```

```
current = self()

spawn_link(fn ->
  for i <- 1..5 do
    send(current, i)
  end
  exit(:noooooo)
end)

for _ <- 1..5 do
  receive do
    msg -> IO.puts(msg)
  end
end
```

```
** (EXIT from #PID<0.107.0>) shell process exited with reason: :noooooo
```

Supervisors

```elixir
defmodule CellClockSupervisor do
    use Supervisor

    def start_link do
        Supervisor.start_link(__MODULE__, :ok)
    end

    def init(_) do
        children = [
            {Clock, [5_000]},
            {Cell, [10_000]}
        ]

        Supervisor.init(children, [strategy: :one_for_one])
    end
end
```

```elixir
defmodule CellClockSupervisor do
    use Supervisor

    def start_link do
        Supervisor.start_link(__MODULE__, :ok)
    end

    def init(_) do
        children = [
            {Clock, [5_000]},
            {Cell, [10_000]}
        ]

        Supervisor.init(children, [strategy: :one_for_one])
    end
end
```

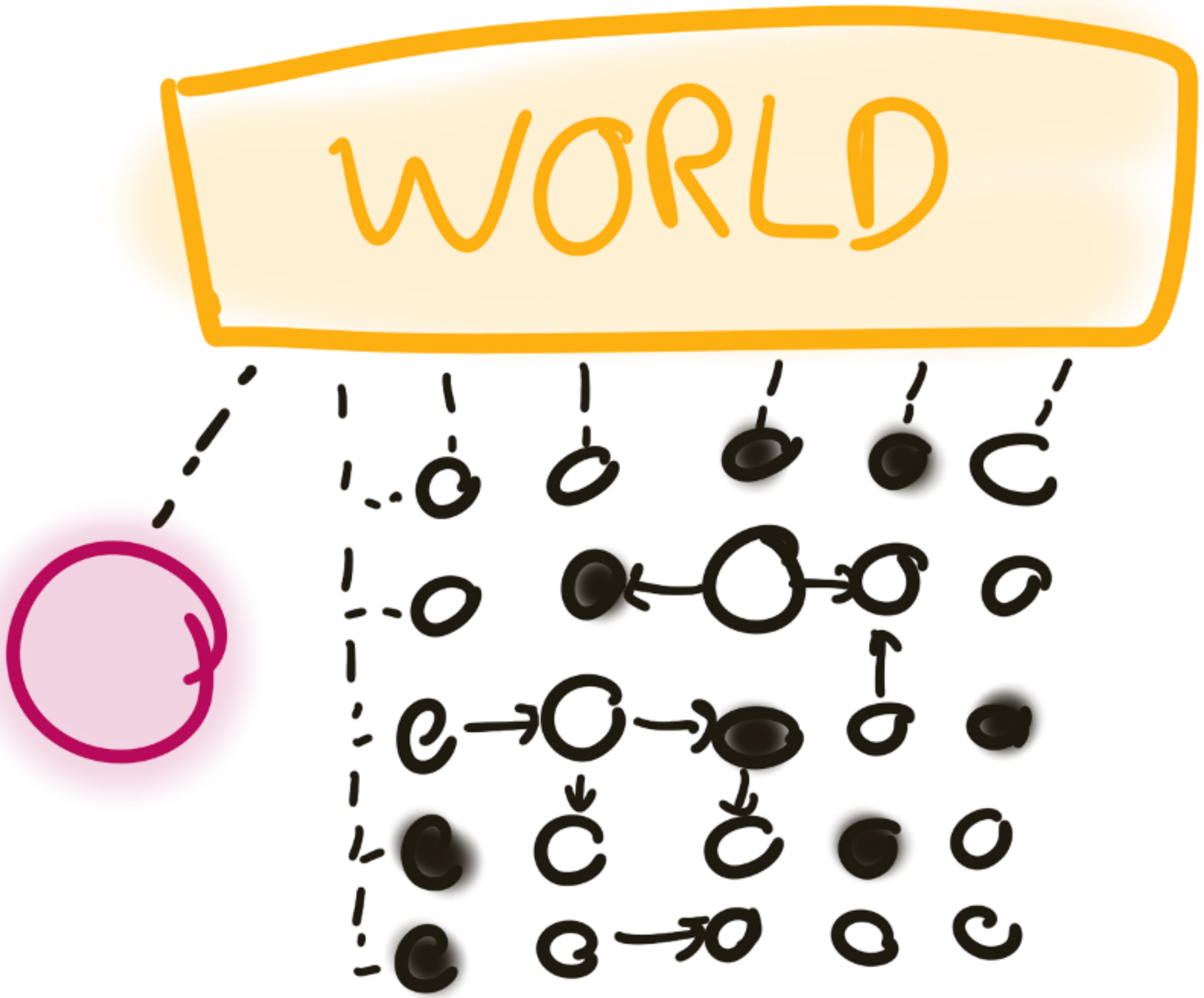
```elixir
defmodule CellClockSupervisor do
    use Supervisor

    def start_link do
        Supervisor.start_link(__MODULE__, :ok)
    end

    def init(_) do
        children = [
            {Clock, [5_000]},
            {Cell, [10_000]}
        ]

        Supervisor.init(children, [strategy: :one_for_one])
    end
end
```

```elixir
defmodule CellClockSupervisor do
    use Supervisor

    def start_link do
        Supervisor.start_link(__MODULE__, :ok)
    end

    def init(_) do
        children = [
            {Clock, [5_000]},
            {Cell, [10_000]}
        ]

        Supervisor.init(children, [strategy: :one_for_one])
    end
end
```

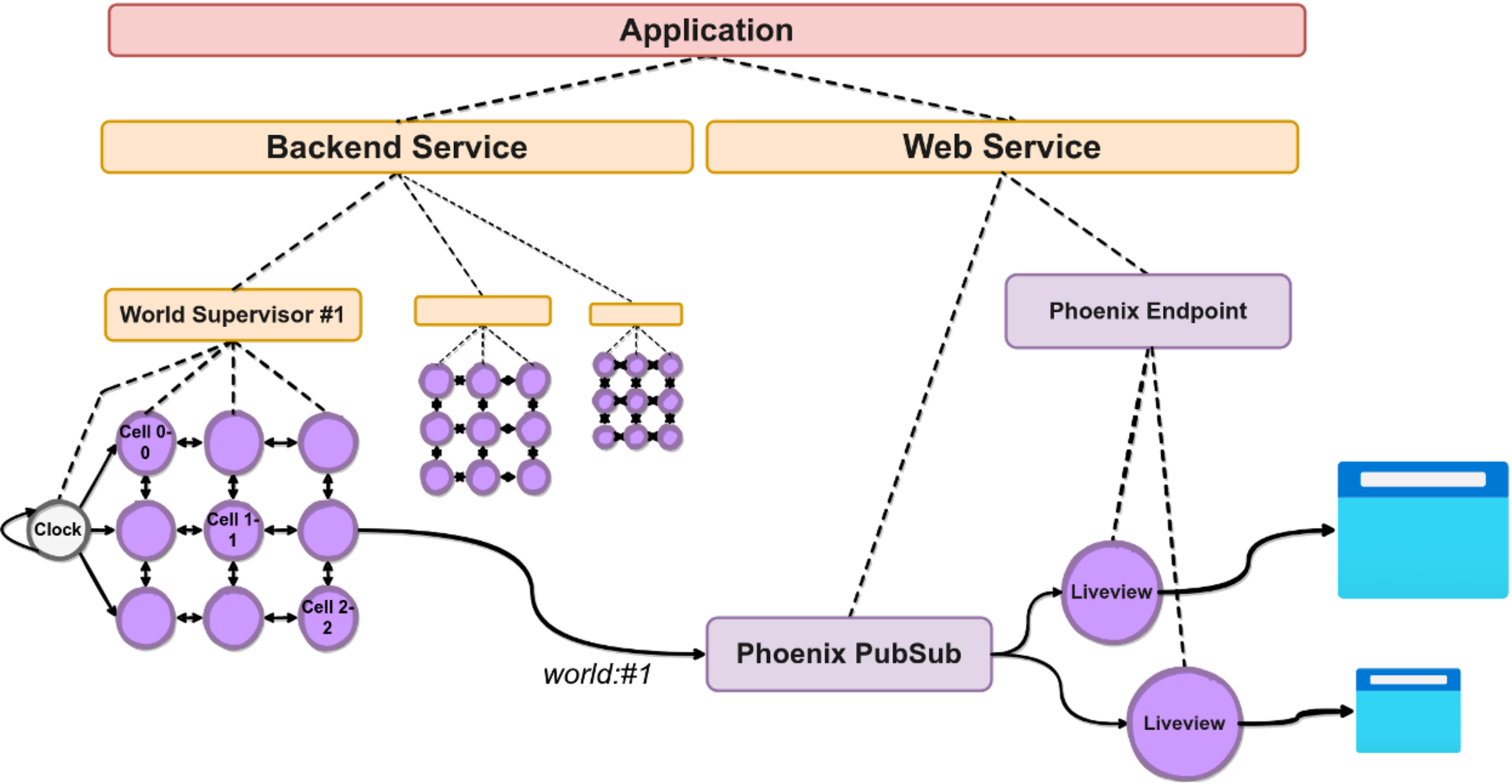
```elixir
CellClockSupervisor.start_link()
```

Let's build an app!

Conway's Game of life


WORLD

Two (micro?)services into one monolith



Backend Service

Events

Events

```elixir
defmodule Gameoflife.Events do
    defmodule Tick do
        defstruct [:t]
    end

    defmodule Tock do
        defstruct [:t]
    end

    defmodule Ping do
        defstruct [:t]
    end

    defmodule On do
        defstruct [:t, :x, :y]
    end

    defmodule Off do
        defstruct [:t, :x, :y]
    end
end
```

Events

```elixir
defmodule Gameoflife.Events do
    defmodule Tick do
        defstruct [:t]
    end

    defmodule Tock do
        defstruct [:t]
    end

    defmodule Ping do
        defstruct [:t]
    end

    defmodule On do
        defstruct [:t, :x, :y]
    end

    defmodule Off do
        defstruct [:t, :x, :y]
    end
end
```

Events

```elixir
defmodule Gameoflife.Events do
    defmodule Tick do
        defstruct [:t]
    end

    defmodule Tock do
        defstruct [:t]
    end

    defmodule Ping do
        defstruct [:t]
    end

    defmodule On do
        defstruct [:t, :x, :y]
    end

    defmodule Off do
        defstruct [:t, :x, :y]
    end
end
```

Events

```elixir
defmodule Gameoflife.Events do
    defmodule Tick do
        defstruct [:t]
    end

    defmodule Tock do
        defstruct [:t]
    end

    defmodule Ping do
        defstruct [:t]
    end

    defmodule On do
        defstruct [:t, :x, :y]
    end

    defmodule Off do
        defstruct [:t, :x, :y]
    end
end
```

```elixir
defmodule Gameoflife.Cell do
  defstruct [:world, :x, :y, :t, :alive?, :neighbors, :failure_rate]

  use GenServer

  alias Gameoflife.Events.{Off, On, Ping, Tick, Tock}

  def start_link(args) do
    GenServer.start_link(__MODULE__, args, name: args[:via])
  end

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

  def broadcast(cell, event) do
    Phoenix.PubSub.broadcast(Gameoflife.PubSub, "world:" <> cell.world.id, event)
  end

  def cast(id, x, y, msg) do
    GenServer.cast({:via, Registry, {Gameoflife.Registry, "cell-#{id}-#{x}-#{y}"}},
  end

```

```elixir
defmodule Gameoflife.Cell do
  defstruct [:world, :x, :y, :t, :alive?, :neighbors, :failure_rate]

  use GenServer

  alias Gameoflife.Events.{Off, On, Ping, Tick, Tock}

  def start_link(args) do
    GenServer.start_link(__MODULE__, args, name: args[:via])
  end

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

  def broadcast(cell, event) do
    Phoenix.PubSub.broadcast(Gameoflife.PubSub, "world:" <> cell.world.id, event)
  end

  def cast(id, x, y, msg) do
    GenServer.cast({:via, Registry, {Gameoflife.Registry, "cell-#{id}-#{x}-#{y}"}},
  end

```

```elixir
defmodule Gameoflife.Cell do
  defstruct [:world, :x, :y, :t, :alive?, :neighbors, :failure_rate]

  use GenServer

  alias Gameoflife.Events.{Off, On, Ping, Tick, Tock}

  def start_link(args) do
    GenServer.start_link(__MODULE__, args, name: args[:via])
  end

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

  def broadcast(cell, event) do
    Phoenix.PubSub.broadcast(Gameoflife.PubSub, "world:" <> cell.world.id, event)
  end

  def cast(id, x, y, msg) do
    GenServer.cast({:via, Registry, {Gameoflife.Registry, "cell-#{id}-#{x}-#{y}"}},
  end

```

```elixir
  def start_link(args) do
    GenServer.start_link(__MODULE__, args, name: args[:via])
  end

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

  def broadcast(cell, event) do
    Phoenix.PubSub.broadcast(Gameoflife.PubSub, "world:" <> cell.world.id, event)
  end

  def cast(id, x, y, msg) do
    GenServer.cast({:via, Registry, {Gameoflife.Registry, "cell-#{id}-#{x}-#{y}"}},
  end

  def handle_cast(%Tick{t: t}, cell) do
    if cell.alive? do
      cast(cell.world.id, cell.x - 1, cell.y - 1, %Ping{t: t})
      cast(cell.world.id, cell.x - 1, cell.y + 0, %Ping{t: t})
      cast(cell.world.id, cell.x - 1, cell.y + 1, %Ping{t: t})
      cast(cell.world.id, cell.x + 0, cell.y - 1, %Ping{t: t})
      cast(cell.world.id, cell.x + 0, cell.y + 1, %Ping{t: t})
```

```elixir
  def cast(id, x, y, msg) do
    GenServer.cast({:via, Registry, {Gameoflife.Registry, "cell-#{id}-#{x}-#{y}"}},
  end

  def handle_cast(%Tick{t: t}, cell) do
    if cell.alive? do
      cast(cell.world.id, cell.x - 1, cell.y - 1, %Ping{t: t})
      cast(cell.world.id, cell.x - 1, cell.y + 0, %Ping{t: t})
      cast(cell.world.id, cell.x - 1, cell.y + 1, %Ping{t: t})
      cast(cell.world.id, cell.x + 0, cell.y - 1, %Ping{t: t})
      cast(cell.world.id, cell.x + 0, cell.y + 1, %Ping{t: t})
      cast(cell.world.id, cell.x + 1, cell.y - 1, %Ping{t: t})
      cast(cell.world.id, cell.x + 1, cell.y + 0, %Ping{t: t})
      cast(cell.world.id, cell.x + 1, cell.y + 1, %Ping{t: t})
    end

    {:noreply, %{cell | t: t}}
  end

  def handle_cast(%Ping{t: t}, cell) do
    {:noreply, %{cell | neighbors: cell.neighbors + 1}}
  end

```

```
29          cast(cell.world.id, cell.x + 0, cell.y - 1, %Ping{t: t})
30          cast(cell.world.id, cell.x + 0, cell.y + 1, %Ping{t: t})
31          cast(cell.world.id, cell.x + 1, cell.y - 1, %Ping{t: t})
32          cast(cell.world.id, cell.x + 1, cell.y + 0, %Ping{t: t})
33          cast(cell.world.id, cell.x + 1, cell.y + 1, %Ping{t: t})
34        end
35
36        {:noreply, %{cell | t: t}}
37      end
38
39      def handle_cast(%Ping{t: t}, cell) do
40        {:noreply, %{cell | neighbors: cell.neighbors + 1}}
41      end
42
43      def handle_cast(%Tock{t: t}, cell) do
44        alive? =
45          case {cell.alive?, cell.neighbors} do
46            {true, 2} -> true
47            {_, 3} -> true
48            _ -> false
49          end
50
51        case {cell.alive?, alive?} do
```

```
37      end
38
39      def handle_cast(%Ping{t: t}, cell) do
40        {:noreply, %{cell | neighbors: cell.neighbors + 1}}
41      end
42
43      def handle_cast(%Tock{t: t}, cell) do
44        alive? =
45          case {cell.alive?, cell.neighbors} do
46            {true, 2} -> true
47            {_, 3} -> true
48            _ -> false
49          end
50
51        case {cell.alive?, alive?} do
52          {false, true} -> broadcast(cell, %On{t: cell.t, x: cell.x, y: cell.y})
53          {true, false} -> broadcast(cell, %Off{t: cell.t, x: cell.x, y: cell.y})
54          _ -> :ok
55        end
56
57        {:noreply, %{cell | t: t, neighbors: 0, alive?: alive?}}
58      end
59  end
```

```
35
36        {:noreply, %{cell | t: t}}
37      end
38
39      def handle_cast(%Ping{t: t}, cell) do
40        {:noreply, %{cell | neighbors: cell.neighbors + 1}}
41      end
42
43      def handle_cast(%Tock{t: t}, cell) do
44        alive? =
45          case {cell.alive?, cell.neighbors} do
46            {true, 2} -> true
47            {_, 3} -> true
48            _ -> false
49          end
50
51        case {cell.alive?, alive?} do
52          {false, true} -> broadcast(cell, %On{t: cell.t, x: cell.x, y: cell.y})
53          {true, false} -> broadcast(cell, %Off{t: cell.t, x: cell.x, y: cell.y})
54          _ -> :ok
55        end
56
57        {:noreply, %{cell | t: t, neighbors: 0, alive?: alive?}}
58      end
```

```
37      end
38
39      def handle_cast(%Ping{t: t}, cell) do
40        {:noreply, %{cell | neighbors: cell.neighbors + 1}}
41      end
42
43      def handle_cast(%Tock{t: t}, cell) do
44        alive? =
45          case {cell.alive?, cell.neighbors} do
46            {true, 2} -> true
47            {_, 3} -> true
48            _ -> false
49          end
50
51        case {cell.alive?, alive?} do
52          {false, true} -> broadcast(cell, %On{t: cell.t, x: cell.x, y: cell.y})
53          {true, false} -> broadcast(cell, %Off{t: cell.t, x: cell.x, y: cell.y})
54          _ -> :ok
55        end
56
57        {:noreply, %{cell | t: t, neighbors: 0, alive?: alive?}}
58      end
59  end
```

```
37      end
38
39      def handle_cast(%Ping{t: t}, cell) do
40        {:noreply, %{cell | neighbors: cell.neighbors + 1}}
41      end
42
43      def handle_cast(%Tock{t: t}, cell) do
44        alive? =
45          case {cell.alive?, cell.neighbors} do
46            {true, 2} -> true
47            {_, 3} -> true
48            _ -> false
49          end
50
51        case {cell.alive?, alive?} do
52          {false, true} -> broadcast(cell, %On{t: cell.t, x: cell.x, y: cell.y})
53          {true, false} -> broadcast(cell, %Off{t: cell.t, x: cell.x, y: cell.y})
54          _ -> :ok
55        end
56
57        {:noreply, %{cell | t: t, neighbors: 0, alive?: alive?}}
58      end
59  end
```

```elixir
defmodule Gameoflife.Clock do
    defstruct id: nil, world: nil, t: 0

    use GenServer

    alias Gameoflife.Events.{Tick, Tock}

    def start_link(args) do
        GenServer.start_link(__MODULE__, args, name: args[:via])
    end

    def init(args) do
        Process.send_after(self(), :tick, 1_000)
        {:ok, args[:clock]}
    end

    def handle_info(:tick, clock) do
        Process.send_after(self(), :tick, 1_000)
        Process.send_after(self(), :tock, 500)

        broadcast(clock, %Tick{t: clock.t + 1})

        {:noreply, %{clock | t: clock.t + 1}}
```

```elixir
defmodule Gameoflife.Clock do
    defstruct id: nil, world: nil, t: 0

    use GenServer

    alias Gameoflife.Events.{Tick, Tock}

    def start_link(args) do
        GenServer.start_link(__MODULE__, args, name: args[:via])
    end

    def init(args) do
        Process.send_after(self(), :tick, 1_000)
        {:ok, args[:clock]}
    end

    def handle_info(:tick, clock) do
        Process.send_after(self(), :tick, 1_000)
        Process.send_after(self(), :tock, 500)

        broadcast(clock, %Tick{t: clock.t + 1})

        {:noreply, %{clock | t: clock.t + 1}}
```

```elixir
defmodule Gameoflife.Clock do
    defstruct id: nil, world: nil, t: 0

    use GenServer

    alias Gameoflife.Events.{Tick, Tock}

    def start_link(args) do
        GenServer.start_link(__MODULE__, args, name: args[:via])
    end

    def init(args) do
        Process.send_after(self(), :tick, 1_000)
        {:ok, args[:clock]}
    end

    def handle_info(:tick, clock) do
        Process.send_after(self(), :tick, 1_000)
        Process.send_after(self(), :tock, 500)

        broadcast(clock, %Tick{t: clock.t + 1})

        {:noreply, %{clock | t: clock.t + 1}}
```

```
 9          GenServer.start_link(__MODULE__, args, name: args[:via])
10      end
11
12      def init(args) do
13          Process.send_after(self(), :tick, 1_000)
14          {:ok, args[:clock]}
15      end
16
17      def handle_info(:tick, clock) do
18          Process.send_after(self(), :tick, 1_000)
19          Process.send_after(self(), :tock, 500)
20
21          broadcast(clock, %Tick{t: clock.t + 1})
22
23          {:noreply, %{clock | t: clock.t + 1}}
24      end
25
26      def handle_info(:tock, clock) do
27          broadcast(clock, %Tock{t: clock.t + 1})
28          {:noreply, clock}
29      end
30
31      defp broadcast(clock, event) do
32          for i <- 0..(clock.world.rows - 1) do
```

```
17      def handle_info(:tick, clock) do
18          Process.send_after(self(), :tick, 1_000)
19          Process.send_after(self(), :tock, 500)
20
21          broadcast(clock, %Tick{t: clock.t + 1})
22
23          {:noreply, %{clock | t: clock.t + 1}}
24      end
25
26      def handle_info(:tock, clock) do
27          broadcast(clock, %Tock{t: clock.t + 1})
28          {:noreply, clock}
29      end
30
31      defp broadcast(clock, event) do
32          for i <- 0..(clock.world.rows - 1) do
33              for j <- 0..(clock.world.columns - 1) do
34                  Gameoflife.Cell.cast(clock.world.id, i, j, event)
35              end
36          end
37
38          Phoenix.PubSub.broadcast(Gameoflife.PubSub, "world:" <> clock.world.id, even
39      end
```

```
18          Process.send_after(self(), :tick, 1_000)
19          Process.send_after(self(), :tock, 500)
20
21          broadcast(clock, %Tick{t: clock.t + 1})
22
23          {:noreply, %{clock | t: clock.t + 1}}
24      end
25
26      def handle_info(:tock, clock) do
27          broadcast(clock, %Tock{t: clock.t + 1})
28          {:noreply, clock}
29      end
30
31      defp broadcast(clock, event) do
32          for i <- 0..(clock.world.rows - 1) do
33              for j <- 0..(clock.world.columns - 1) do
34                  Gameoflife.Cell.cast(clock.world.id, i, j, event)
35              end
36          end
37
38          Phoenix.PubSub.broadcast(Gameoflife.PubSub, "world:" <> clock.world.id, even
39      end
40  end
```

```elixir
defmodule Gameoflife.World do
  defstruct [:id, :columns, :rows]

  use Supervisor

  alias Gameoflife.{Cell, Clock, World}

  def start_link(args) do
    Supervisor.start_link(__MODULE__, args)
  end

  @impl true
  def init(args) do
    Supervisor.init(args[:children], strategy: :one_for_one)
  end

  def new(columns, rows) do
    world = %World{id: id(4), columns: columns, rows: rows}
    {:ok, pid} =
      DynamicSupervisor.start_child(
        Gameoflife.BackendService,
        {__MODULE__,
         [
```

```elixir
defmodule Gameoflife.World do
  defstruct [:id, :columns, :rows]

  use Supervisor

  alias Gameoflife.{Cell, Clock, World}

  def start_link(args) do
    Supervisor.start_link(__MODULE__, args)
  end

  @impl true
  def init(args) do
    Supervisor.init(args[:children], strategy: :one_for_one)
  end

  def new(columns, rows) do
    world = %World{id: id(4), columns: columns, rows: rows}
    {:ok, pid} =
      DynamicSupervisor.start_child(
        Gameoflife.BackendService,
        {__MODULE__,
         [
```

```elixir
  @impl true
  def init(args) do
    Supervisor.init(args[:children], strategy: :one_for_one)
  end

  def new(columns, rows) do
    world = %World{id: id(4), columns: columns, rows: rows}
    {:ok, pid} =
      DynamicSupervisor.start_child(
        Gameoflife.BackendService,
        {__MODULE__,
        [
          id: id,
          children: cells(world) ++ clock(world)
        ]}
      )
  end

  def cells(%World{rows: n, columns: m} = world) do
    ...
  end

  def clock(%World{id: id} = world) do
```

```
15    end
16
17    def new(columns, rows) do
18      world = %World{id: id(4), columns: columns, rows: rows}
19      {:ok, pid} =
20        DynamicSupervisor.start_child(
21          Gameoflife.BackendService,
22          {__MODULE__,
23           [
24             id: id,
25             children: cells(world) ++ clock(world)
26           ]}
27        )
28    end
29
30    def cells(%World{rows: n, columns: m} = world) do
31        ...
32    end
33
34    def clock(%World{id: id} = world) do
35      ...
36    end
37  end
```

Web Service

```elixir
defmodule Gameoflife.WebService.WorldLive do
  use Gameoflife.WebService, :live_view

  alias Gameoflife.Events.{Off, On, Tick, Tock}

  def mount(%{"id" => id}, _args, socket) do
    Phoenix.PubSub.subscribe(Gameoflife.PubSub, "world:#{id}")
    {:ok, assign(socket, t: nil, id: id, world: world, grid: %{}, buffer: %{})}
  end

  def handle_info(%On{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :on)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Off{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :off)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Tock{t: t}, %{assigns: %{grid: grid, buffer: buffer}} = socket) do
    {:noreply, assign(socket, t: t, grid: Map.merge(grid, buffer), buffer: %{})}
  end
```

```elixir
defmodule Gameoflife.WebService.WorldLive do
  use Gameoflife.WebService, :live_view

  alias Gameoflife.Events.{Off, On, Tick, Tock}

  def mount(%{"id" => id}, _args, socket) do
    Phoenix.PubSub.subscribe(Gameoflife.PubSub, "world:#{id}")
    {:ok, assign(socket, t: nil, id: id, world: world, grid: %{}, buffer: %{})}
  end

  def handle_info(%On{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :on)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Off{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :off)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Tock{t: t}, %{assigns: %{grid: grid, buffer: buffer}} = socket) do
    {:noreply, assign(socket, t: t, grid: Map.merge(grid, buffer), buffer: %{})}
  end
```

```elixir
defmodule Gameoflife.WebService.WorldLive do
  use Gameoflife.WebService, :live_view

  alias Gameoflife.Events.{Off, On, Tick, Tock}

  def mount(%{"id" => id}, _args, socket) do
    Phoenix.PubSub.subscribe(Gameoflife.PubSub, "world:#{id}")
    {:ok, assign(socket, t: nil, id: id, world: world, grid: %{}, buffer: %{})}
  end

  def handle_info(%On{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :on)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Off{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :off)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Tock{t: t}, %{assigns: %{grid: grid, buffer: buffer}} = socket) do
    {:noreply, assign(socket, t: t, grid: Map.merge(grid, buffer), buffer: %{})}
  end
```

```elixir
defmodule Gameoflife.WebService.WorldLive do
  use Gameoflife.WebService, :live_view

  alias Gameoflife.Events.{Off, On, Tick, Tock}

  def mount(%{"id" => id}, _args, socket) do
    Phoenix.PubSub.subscribe(Gameoflife.PubSub, "world:#{id}")
    {:ok, assign(socket, t: nil, id: id, world: world, grid: %{}, buffer: %{})}
  end

  def handle_info(%On{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :on)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Off{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :off)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Tock{t: t}, %{assigns: %{grid: grid, buffer: buffer}} = socket) do
    {:noreply, assign(socket, t: t, grid: Map.merge(grid, buffer), buffer: %{})}
  end
end
```

```elixir
  use Gameoflife.WebService, :live_view

  alias Gameoflife.Events.{Off, On, Tick, Tock}

  def mount(%{"id" => id}, _args, socket) do
    Phoenix.PubSub.subscribe(Gameoflife.PubSub, "world:#{id}")
    {:ok, assign(socket, t: nil, id: id, world: world, grid: %{}, buffer: %{})}
  end

  def handle_info(%On{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :on)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Off{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :off)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Tock{t: t}, %{assigns: %{grid: grid, buffer: buffer}} = socket) do
    {:noreply, assign(socket, t: t, grid: Map.merge(grid, buffer), buffer: %{})}
  end
end
```

```elixir
  use Gameoflife.WebService, :live_view

  alias Gameoflife.Events.{Off, On, Tick, Tock}

  def mount(%{"id" => id}, _args, socket) do
    Phoenix.PubSub.subscribe(Gameoflife.PubSub, "world:#{id}")
    {:ok, assign(socket, t: nil, id: id, world: world, grid: %{}, buffer: %{})}
  end

  def handle_info(%On{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :on)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Off{x: x, y: y}, %{assigns: %{buffer: buffer}} = socket) do
    buffer = Map.put(buffer, {x, y}, :off)
    {:noreply, assign(socket, buffer: buffer)}
  end

  def handle_info(%Tock{t: t}, %{assigns: %{grid: grid, buffer: buffer}} = socket) do
    {:noreply, assign(socket, t: t, grid: Map.merge(grid, buffer), buffer: %{})}
  end
end
```

```
<div class="mt-5 max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">
  <div class="grid grid-cols-#{@world.rows} mt-4 border">

    <%= for i <- 0..(@world.rows - 1) do %>
      <%= for j <- 0..(@world.columns - 1) do %>
        <div class={Map.get(@grid, {i, j})}></div>
      <% end %>
    <% end %>

  </div>
</div>
```

```
<div class="mt-5 max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">
  <div class="grid grid-cols-#{@world.rows} mt-4 border">

    <%= for i <- 0..(@world.rows - 1) do %>
      <%= for j <- 0..(@world.columns - 1) do %>
        <div class={Map.get(@grid, {i, j})}></div>
      <% end %>
    <% end %>

  </div>
</div>
```

```
<div class="mt-5 max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">
  <div class="grid grid-cols-#{@world.rows} mt-4 border">

    <%= for i <- 0..(@world.rows - 1) do %>
      <%= for j <- 0..(@world.columns - 1) do %>
        <div class={Map.get(@grid, {i, j})}></div>
      <% end %>
    <% end %>

  </div>
</div>
```

(Micro)services & Application

```elixir
defmodule Gameoflife.BackendService do
  use DynamicSupervisor

  def start_link(args) do
    Supervisor.start_link(__MODULE__, args)
  end

  def init(_args) do
    DynamicSupervisor.init(strategy: :one_for_one)
  end
end
```

```elixir
defmodule Gameoflife.BackendService do
  use DynamicSupervisor

  def start_link(args) do
    Supervisor.start_link(__MODULE__, args)
  end

  def init(_args) do
    DynamicSupervisor.init(strategy: :one_for_one)
  end
end
```

```elixir
defmodule Gameoflife.WebService do
  use Supervisor

  @impl true
  def init(_init_arg) do
    children = [
      {Phoenix.PubSub, name: Gameoflife.PubSub},
      Gameoflife.WebService.Endpoint
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

```elixir
defmodule Gameoflife.WebService do
  use Supervisor

  @impl true
  def init(_init_arg) do
    children = [
      {Phoenix.PubSub, name: Gameoflife.PubSub},
      Gameoflife.WebService.Endpoint
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

```elixir
defmodule Gameoflife.Application do
    use Application

    def start(_type, _args) do
      children = [
        Gameoflife.WebService,
        Gameoflife.BackendService,
      ]

      Supervisor.start_link(children, strategy: :one_for_one)
    end
end
```

```elixir
defmodule Gameoflife.Application do
    use Application

    def start(_type, _args) do
      children = [
        Gameoflife.WebService,
        Gameoflife.BackendService,
      ]

      Supervisor.start_link(children, strategy: :one_for_one)
    end
end
```

```elixir
defmodule Gameoflife.MixProject do
    use Mix.Project

    def project do
      [
        app: :gameoflife,
        version: "0.1.0",
        elixir: "~> 1.12",
        deps: deps()
      ]
    end

    def application do
      [
        mod: {Gameoflife.Application, []}
      ]
    end

    defp deps do
      [
        {:phoenix, "~> 1.6.6"},
        ...
      ]
```

```elixir

  def project do
    [
      app: :gameoflife,
      version: "0.1.0",
      elixir: "~> 1.12",
      deps: deps()
    ]
  end

  def application do
    [
      mod: {Gameoflife.Application, []}
    ]
  end

  defp deps do
    [
      {:phoenix, "~> 1.6.6"},
      ...
    ]
  end
end
```

mix deps.get

```
mix deps.get
mix release
```

```
mix deps.get
mix release
./gameoflife start
```

```
mix deps.get
mix release
./gameoflife start
```

https://gameoflife.fly.dev

General / Fly App

2023-01-25 17:34:58 to 2023-01-25 17:40:59

App gameoflife

Fly Edge | Fly Instance

Overview

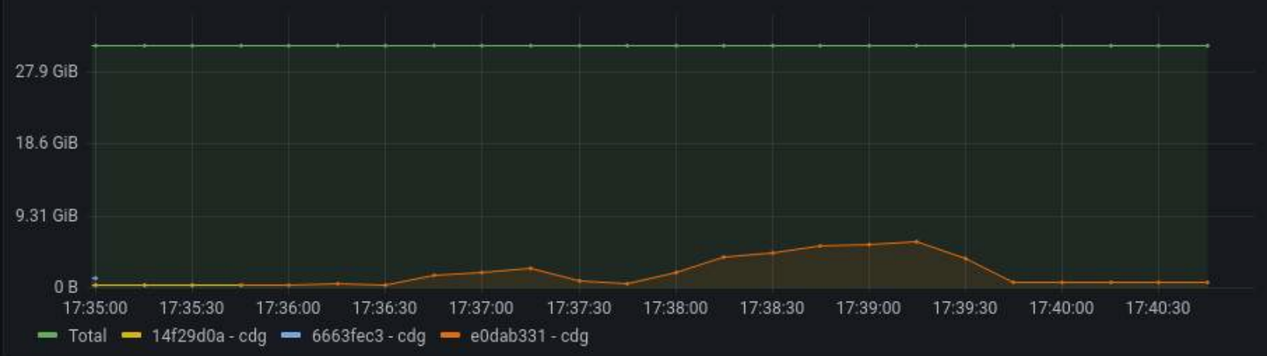
Data Out

Data Out

region	Edge ↓	Instance ↓
cdg		131 kB/s
fra	101 kB/s	
Total	101 kB/s	131 kB/s

Network I/O

Memory Utilization



CPU Utilization

Phoenix Framework

ecto

BROADWAY
AN ELIXIR LIBRARY

OBAN

Nx

Livebook

NERVES

I 💜 Elixir

It works like my 🌎

Elixir Homepage

Elixir School

Elixir in Action (Sasa Juric)

Merci à nos sponsors

VISEO
POSITIVE DIGITAL MAKERS

open
WE EMPOWER
YOUR DIGITAL WORLD

CRITEO

gravitee.io

sopra
steria

zenika

THALES
Building a future we can all trust

ABYLSEN

Microsoft

salesforce

CGI

OVHcloud

AVISTO

diabeloop

LIGHT UP
YOUR FUTURE_
HARDIS
GROUP

KLSGROUP
La French Logistique

kelkoogroup

Moody's

Questions ?

https://gameoflife.fly.dev

Twitter: @maximejanvier