

Intro to Elixir



Table of contents

- Introduction
- Functional programming
- Elixir
- Phoenix
- Elixir & Phoenix in industry
- Demo

Who are we?



Luc Tielen
@luctielen



Hans Kenis
@HansKenis





Kabisa is een maatwerk software ontwikkelaar die zich specialiseert op het gebied van web en mobiele app ontwikkeling & integratie.



nrc carrière > held
2016/2017

10+

jaar in business

300+

succesvolle projecten

40

team members

3

kantoren in NL



Functional programming

- “Programming with functions”
- All about the data
- Functions: transformations of data

Functional programming

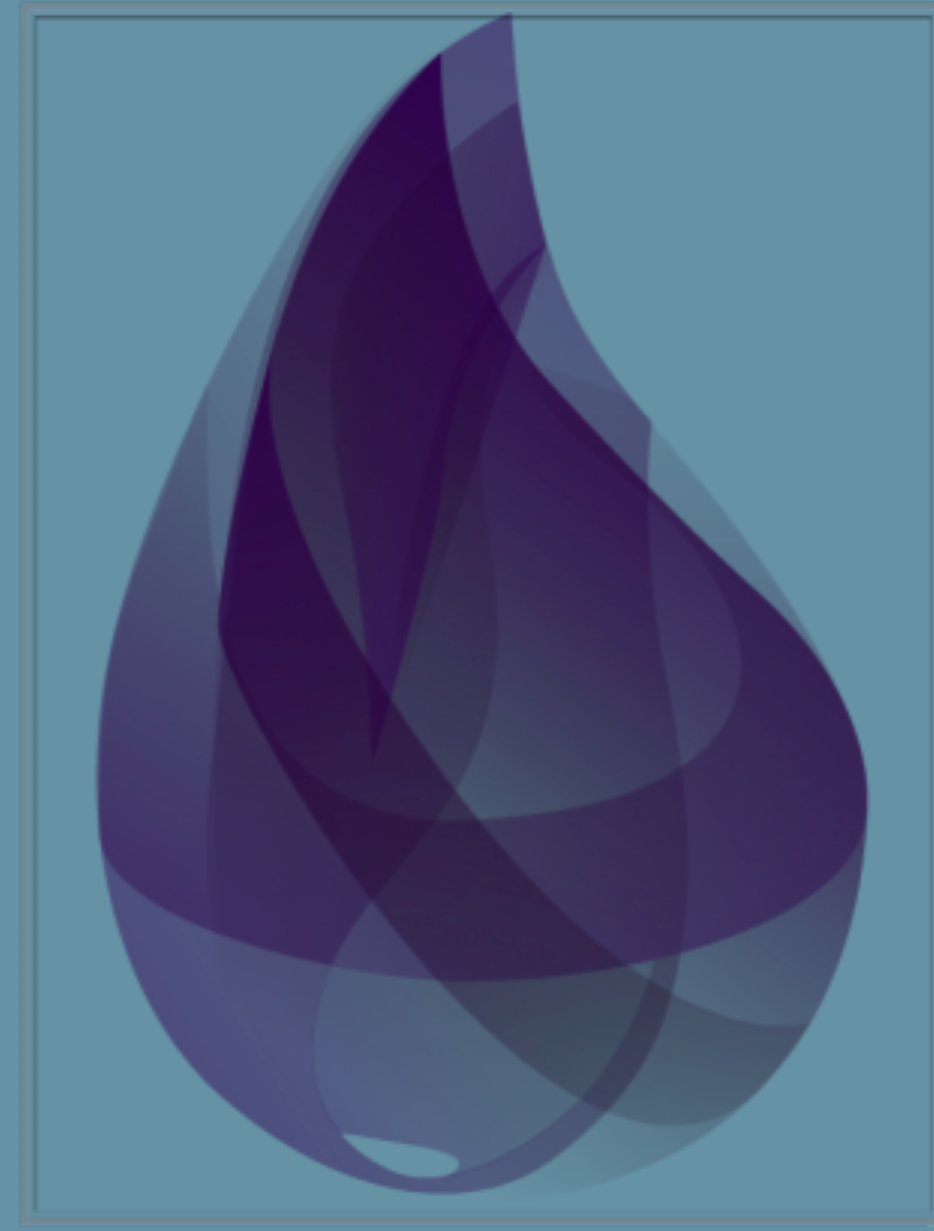
- High level, declarative
- Immutability -> easy to parallelize
- None or minimal side effects
- Same inputs -> same outputs

```
x = 0

def update_counter():
    x += 100

update_counter()
print('x = now %d' % x)

update_counter()
print('x = now %d' % x)
```



“A dynamic, functional language designed for building scalable and maintainable applications”

Elixir

- Functional programming language
- Ruby-like syntax
- Built on top of Erlang
- Macro's (meta-programming)
- Great tooling



A trip back in history: Erlang

- Created by Ericsson in 1986
- Originally designed for telecom industry
- Now: similar to structure of the web



Erlang

- For systems that need:
 - Distributed
 - Fault-tolerant
 - Soft real-time
 - Highly available
 - Hot swapping code



Elixir

- Compiles to Erlang BEAM File
- Runs on Erlang VM
- Erlang compatible

Elixir Crash Course

```
def add(x, y) do  
  x + y  
end
```

- Function definition:
 - Identified by name and arity: `add/2`
 - Dynamic typing
 - Always return last expression (no return statement)

Elixir Crash Course

```
1 defmodule Calculator do
2   def add(x, y) do
3     x + y
4   end
5
6   def subtract(x, y) do
7     x - y
8   end
9
10  def multiply(x, y) do
11    x * y
12  end
13
14  def divide(x, y) do
15    x / y
16  end
17 end
```

- Module:
 - Group of functions

Elixir Crash Course

```
iex(2)> Calculator.add(1, 2)
3
iex(3)> Calculator.subtract(10.23, 1/4)
9.98
iex(4)> Calculator.multiply(2, 8)
16
iex(5)> 
```

- Example usage

Elixir Crash Course

```
1 defmodule ComplexCalculation do
2   import Calculator
3   # (x/2) * 3 - 4 + 5
4   def compute(x) do
5     add(subtract(multiply(divide(x, 2), 3), 4), 5)
6   end
7 end
```

Can be rewritten as

Elixir Crash Course

```
1 defmodule ComplexCalculation do
2   import Calculator
3   # (x/2) * 3 - 4 + 5
4   def compute(x) do
5     x
6     |> divide(2)
7     |> multiply(3)
8     |> subtract(4)
9     |> add(5)
10  end
11 end
12
13 |
```

- Pipe-operator:
 - `|>`
 - `fun_1 |> fun_2`
 - Result of `fun_1` becomes first argument of `fun_2`

Elixir Crash Course

```
1 defmodule Fibonacci do
2   def fib(0), do: 0
3   def fib(1), do: 1
4   def fib(n), do: fib(n-1) + fib(n-2)
5 end
```

- Pattern-matching
- Recursion

Actor model

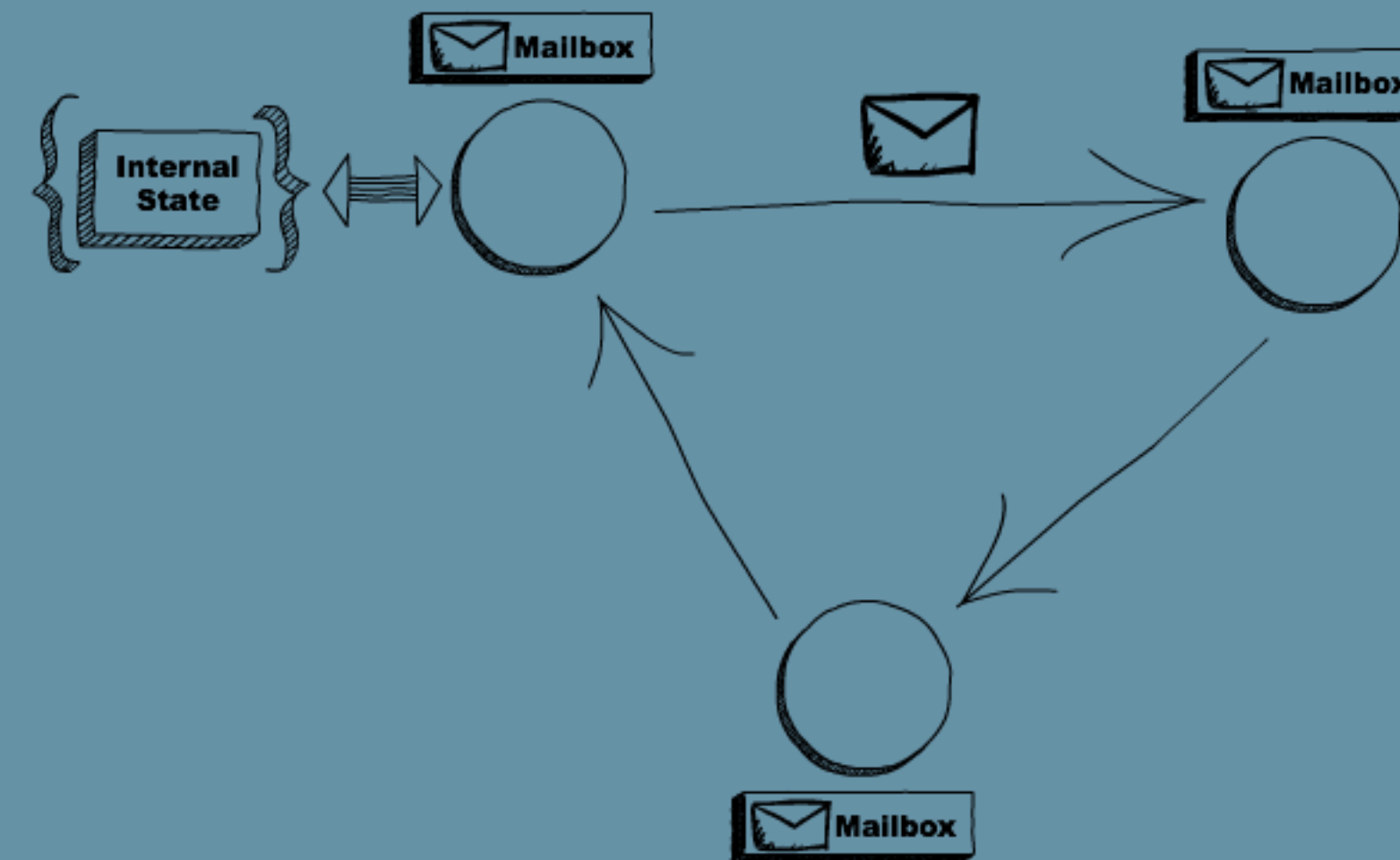
- An actor is a primitive unit of computation:
 1. Processing
 2. Storage
 3. Communication

Actor model

- Actor can (in response to a message):
 1. Send messages to other actors
 2. Create new actors
 3. Change internal state

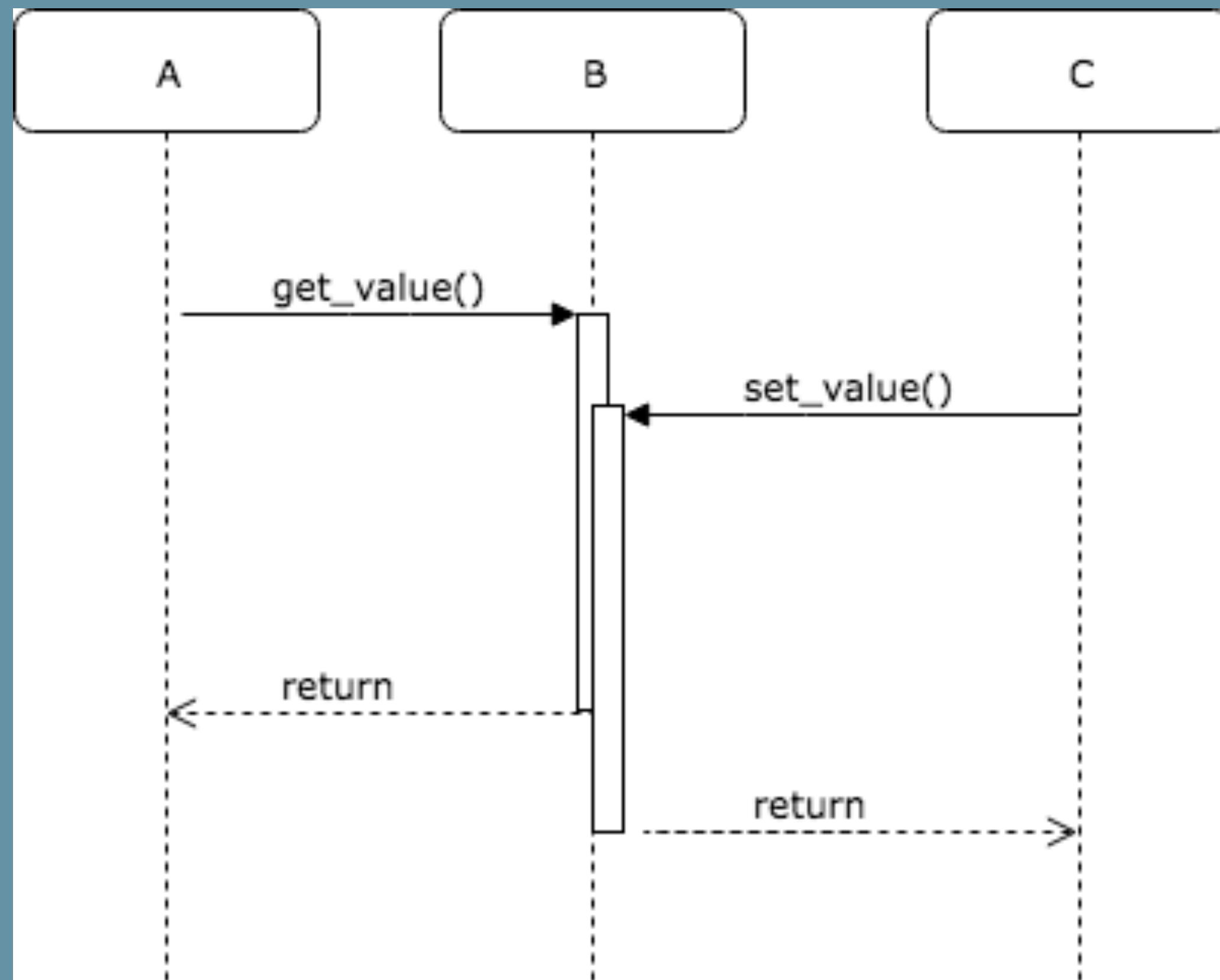
Actor model

- “One Ant is no Ant”
- Actor has an address (mailbox)
- Isolated
- Message passing
- Messages handled sequentially
- Concurrency through multiple actors

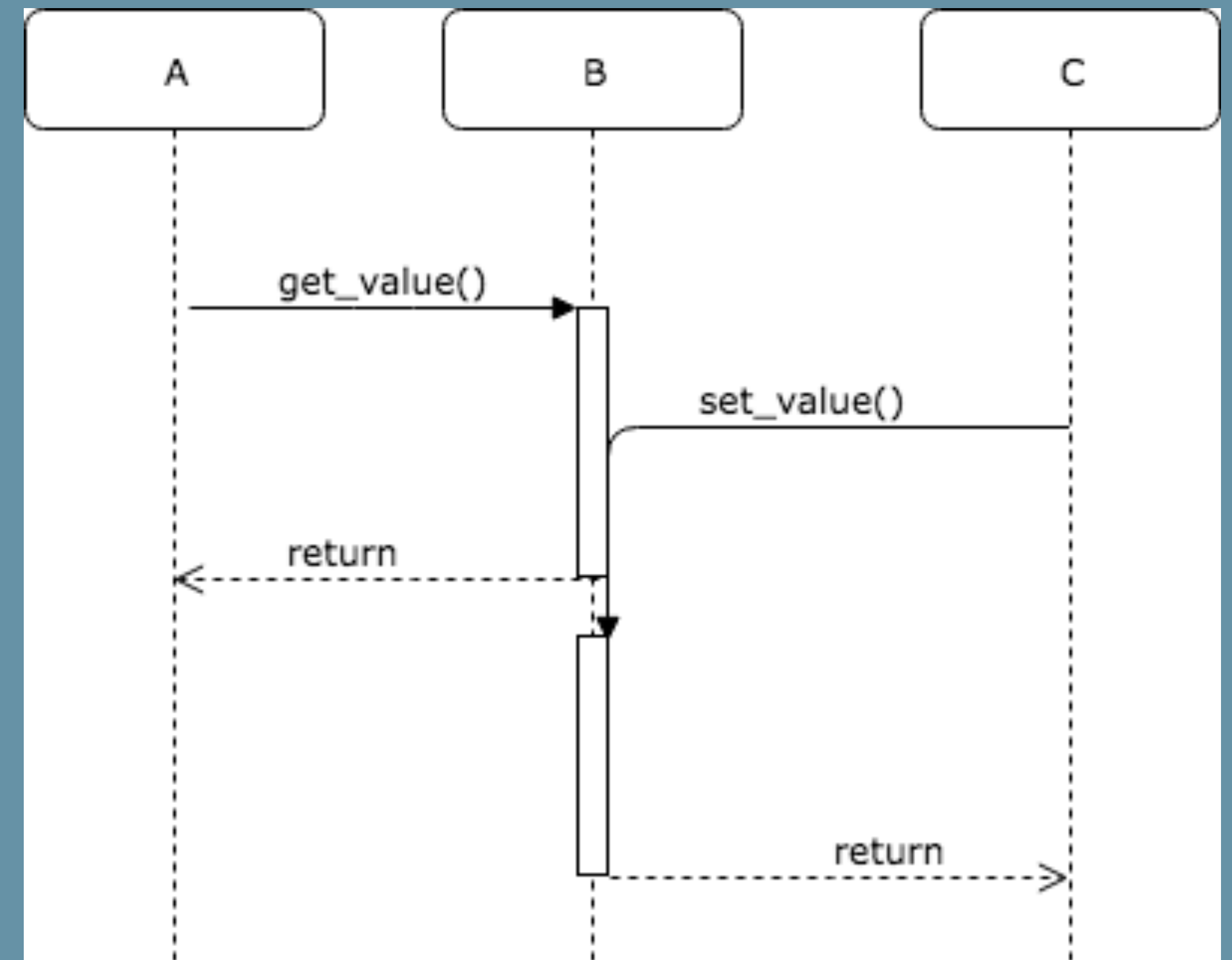


Actor model

Multi-threading in OO



Actor Model



Actor model

- Simple concept -> less complexity
- Less error prone (deadlocks, data races) than mutexes / condition variables / ...
- But not easy to get right when designing a language



Actor model in Elixir

- “Actor” == “process”
- Extremely Lightweight: easily 1M+ on standard HW
- 1 mailbox per process
- Only async messages
- All code runs in processes

Processes: example 1

```
defmodule PrintCounter do  
  def start do  
    spawn(fn -> counter_handler(0) end)  
  end  
end
```

- spawn: creates new process
- returns the process ID (PID) to communicate with process

Processes: example 1

```
def increment(ctr_pid) do
  ctr_pid |> send(:increment)
end
```

- send: sends message to a process

Processes: example 1

```
def counter_handler(counter) do
  receive do
    :increment ->
      counter_updated = counter + 1
      IO.puts "Counter is now #{counter_updated}!"
      counter_handler(counter_updated)
  end
end
end
```

- receive: listen for messages
- explicit updating of state

Processes: example 1

```
defmodule PrintCounter do

  def start do
    spawn(fn -> counter_handler(0) end)
  end

  def increment(ctr_pid) do
    ctr_pid |> send(:increment)
  end

  def counter_handler(counter) do
    receive do
      :increment ->
        counter_updated = counter + 1
        IO.puts "Counter is now #{counter_updated}!"
        counter_handler(counter_updated)
    end
  end
end
```

Processes: example 2

```
def get_value(ctr_pid) do
  ctr_pid |> send({:get_value, self()})
  receive do
    {:value, value} -> value
  end
end
```

- Synchronous message = async message back and forth
- self() returns current PID

Processes: example 2

```
def counter_handler(counter) do
  receive do
    :increment ->
      counter_handler(counter + 1)
    {:get_value, pid} ->
      pid |> send({:value, counter})
      counter_handler(counter)
  end
end
```

- receiving process updated to send message back to sender

Processes: example 2

```
defmodule Counter do
  def start, do: spawn(fn -> counter_handler(0) end)

  def increment(ctr_pid), do: ctr_pid |> send(:increment)

  def get_value(ctr_pid) do
    ctr_pid |> send({:get_value, self()})
    receive do
      {:value, value} -> value
    end
  end
end

def counter_handler(counter) do
  receive do
    :increment ->
      counter_handler(counter + 1)
    {:get_value, pid} ->
      pid |> send({:value, counter})
      counter_handler(counter)
  end
end
end
```

Processes: example 2

```
Interactive Elixir (1.6.4) - press Ctrl+C to exit
iex(1)> pid = Counter.start
#PID<0.111.0>
iex(2)> Counter.increment(pid)
:increment
iex(3)> Counter.increment(pid)
:increment
iex(4)> Counter.increment(pid)
:increment
iex(5)> Counter.get_value(pid)
3
```

OTP

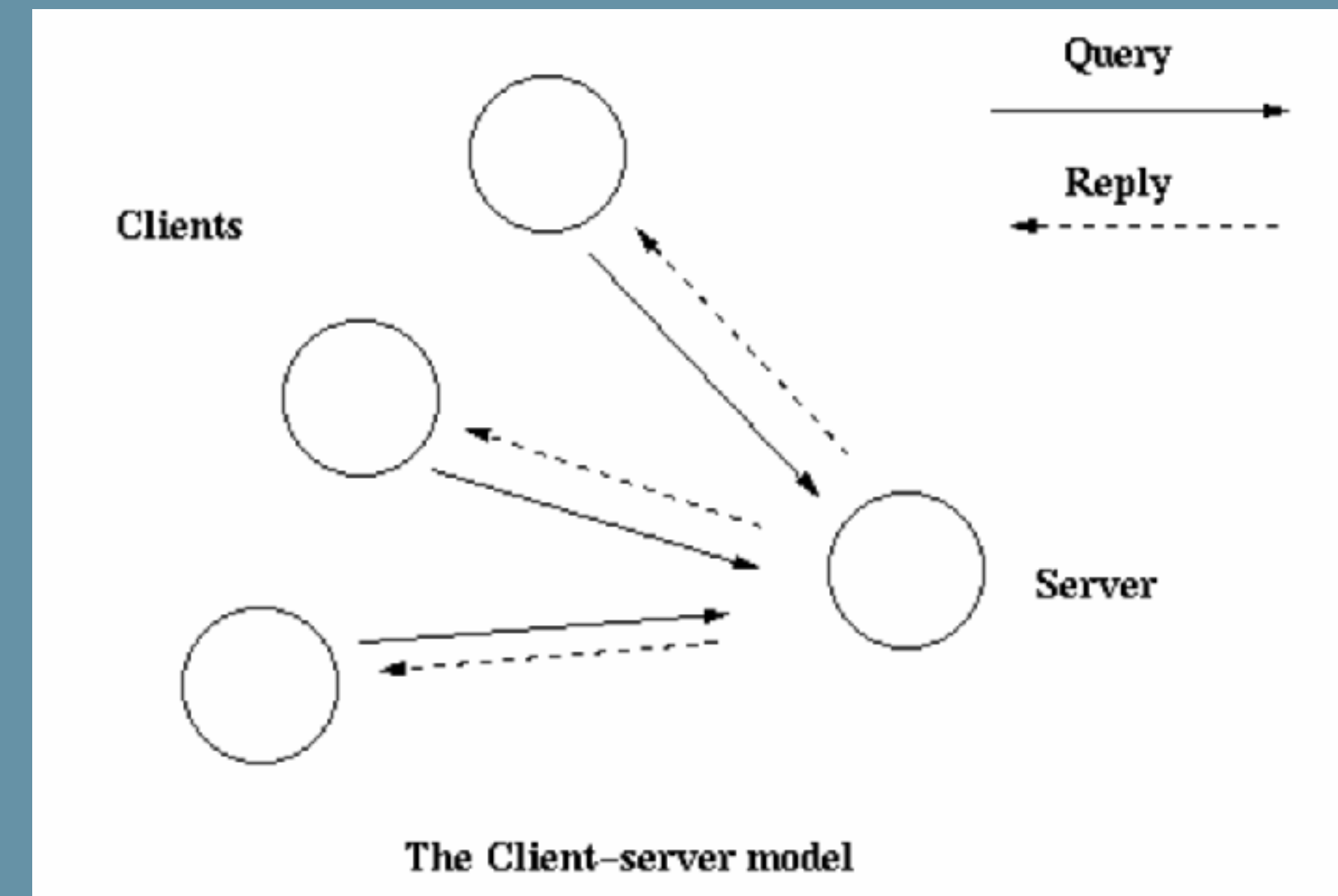
- "Open Telecom Platform"
- Middleware
- Abstracts away boilerplate
- Battle hardened for over 30 years
- Let it crash / fail fast mentality



**KEEP
CALM
AND
LET IT
CRASH**

OTP: GenServer

- = Generic Server
- Abstraction layer on top of processes
- Hooks for:
 - Initialization/termination of process
 - Message handling (async, sync, out of band)
 - Hot code swapping



OTP: GenServer example

- Keep stateful code separated as much as possible!

```
defmodule CounterServer do
  use GenServer

  # 'Client' side:

  def start_link, do: GenServer.start_link(__MODULE__, 0)

  def increment(pid), do: GenServer.cast(pid, :increment)

  def get_value(pid), do: GenServer.call(pid, :get_value)

  # 'Server' side:

  def init(value), do: {:ok, value}

  def handle_cast(:increment, value) do
    {:noreply, value + 1}
  end

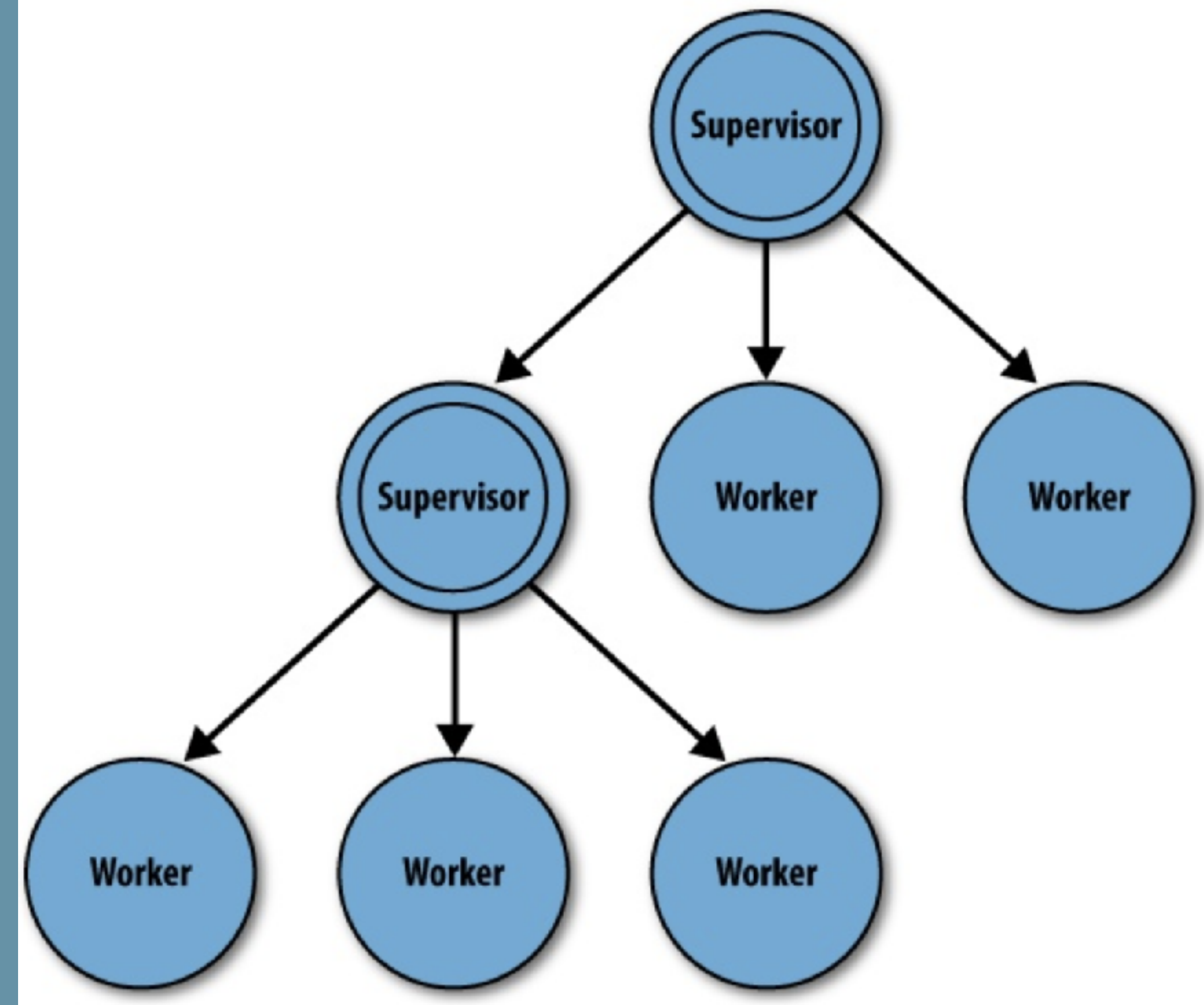
  def handle_call(:get_value, _from, value) do
    response = value
    {:reply, response, value}
  end
end
```

OTP: GenServer example

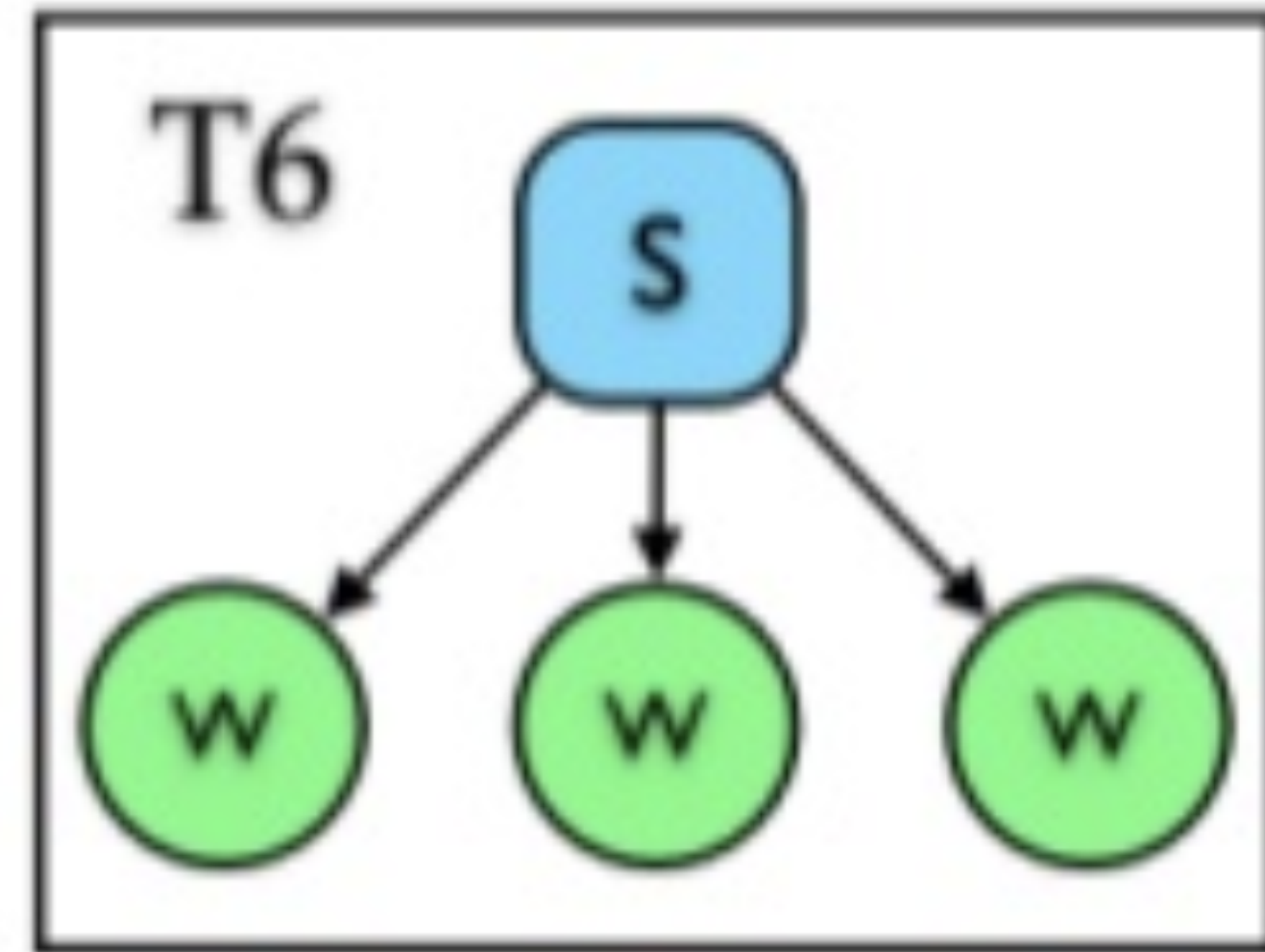
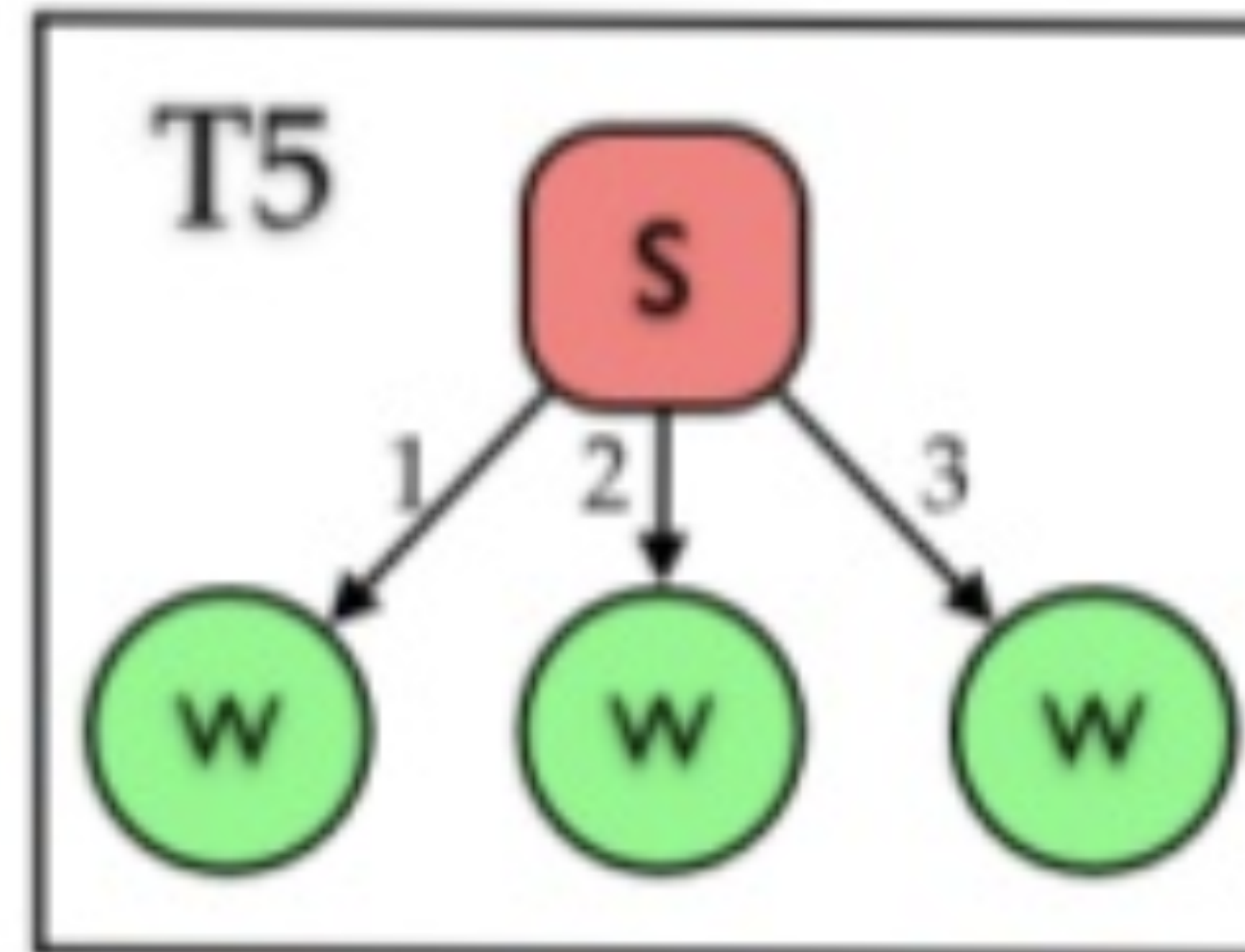
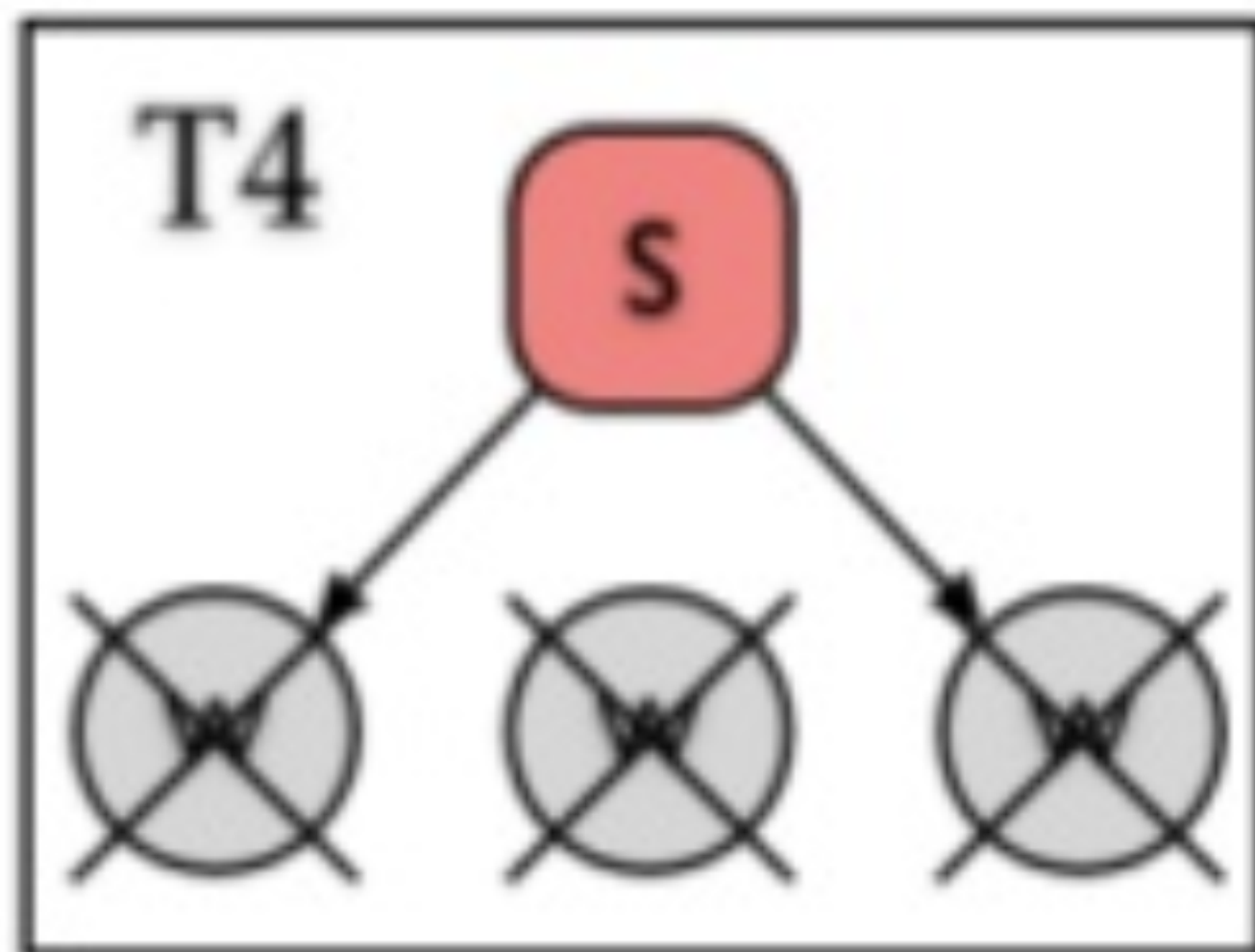
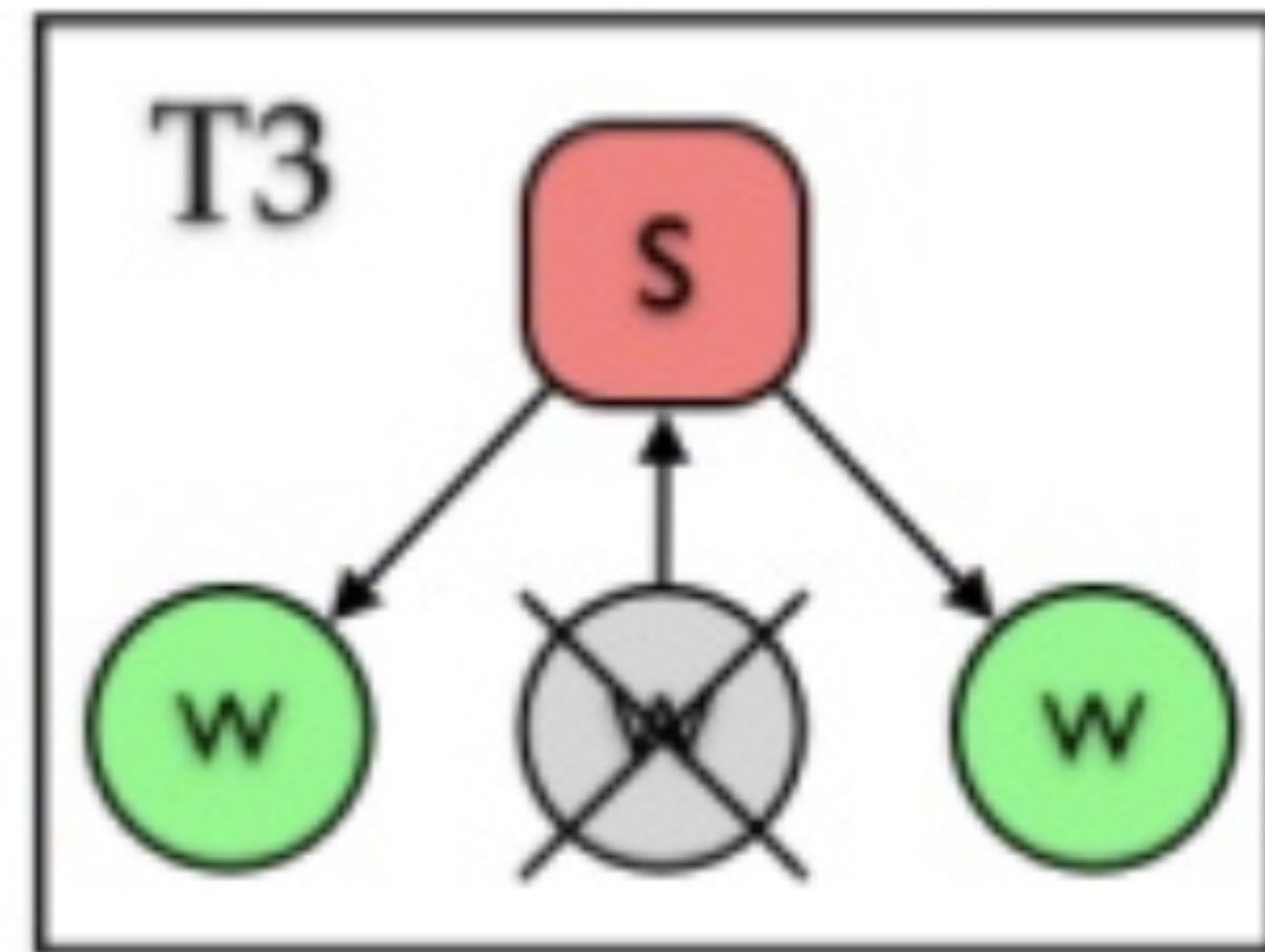
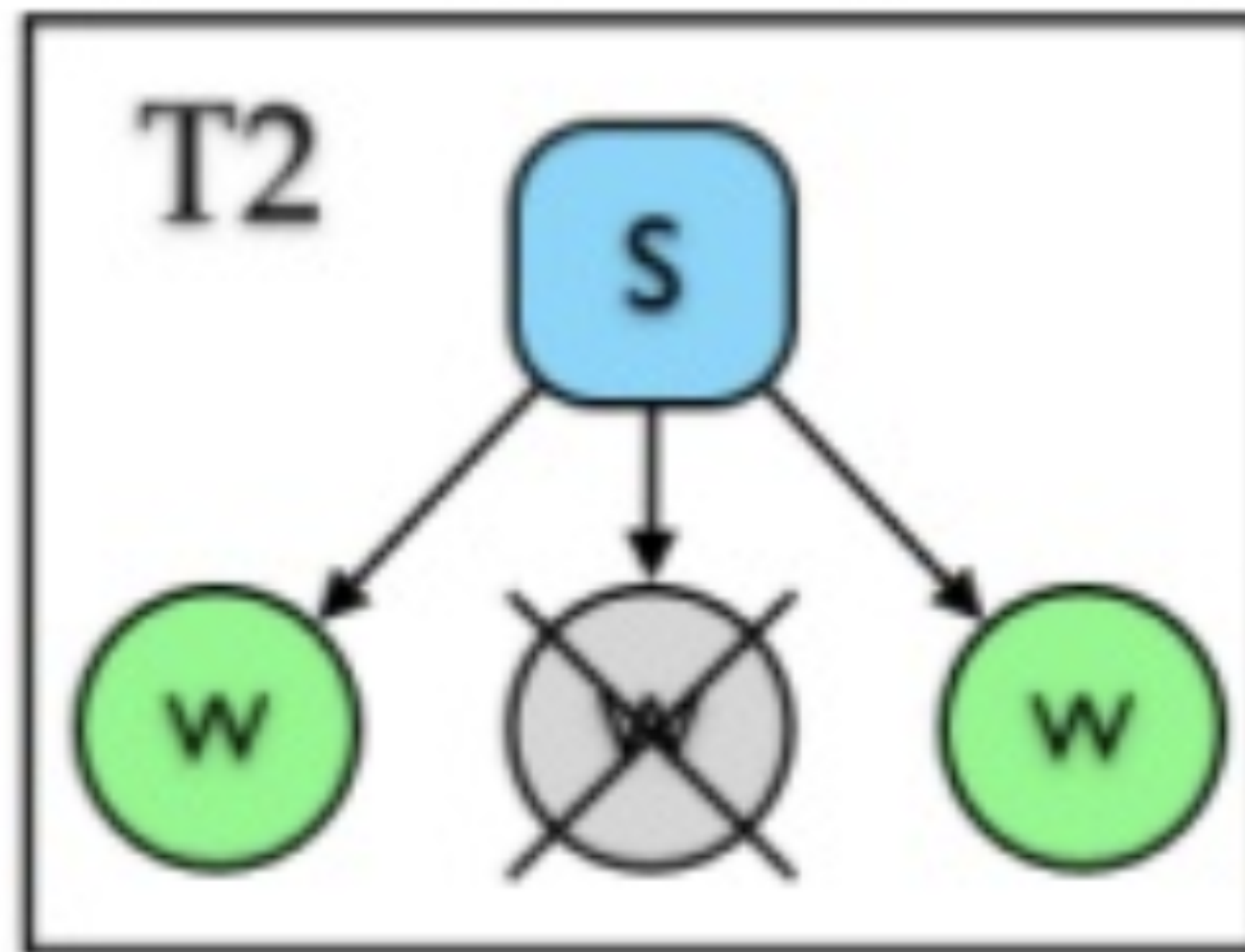
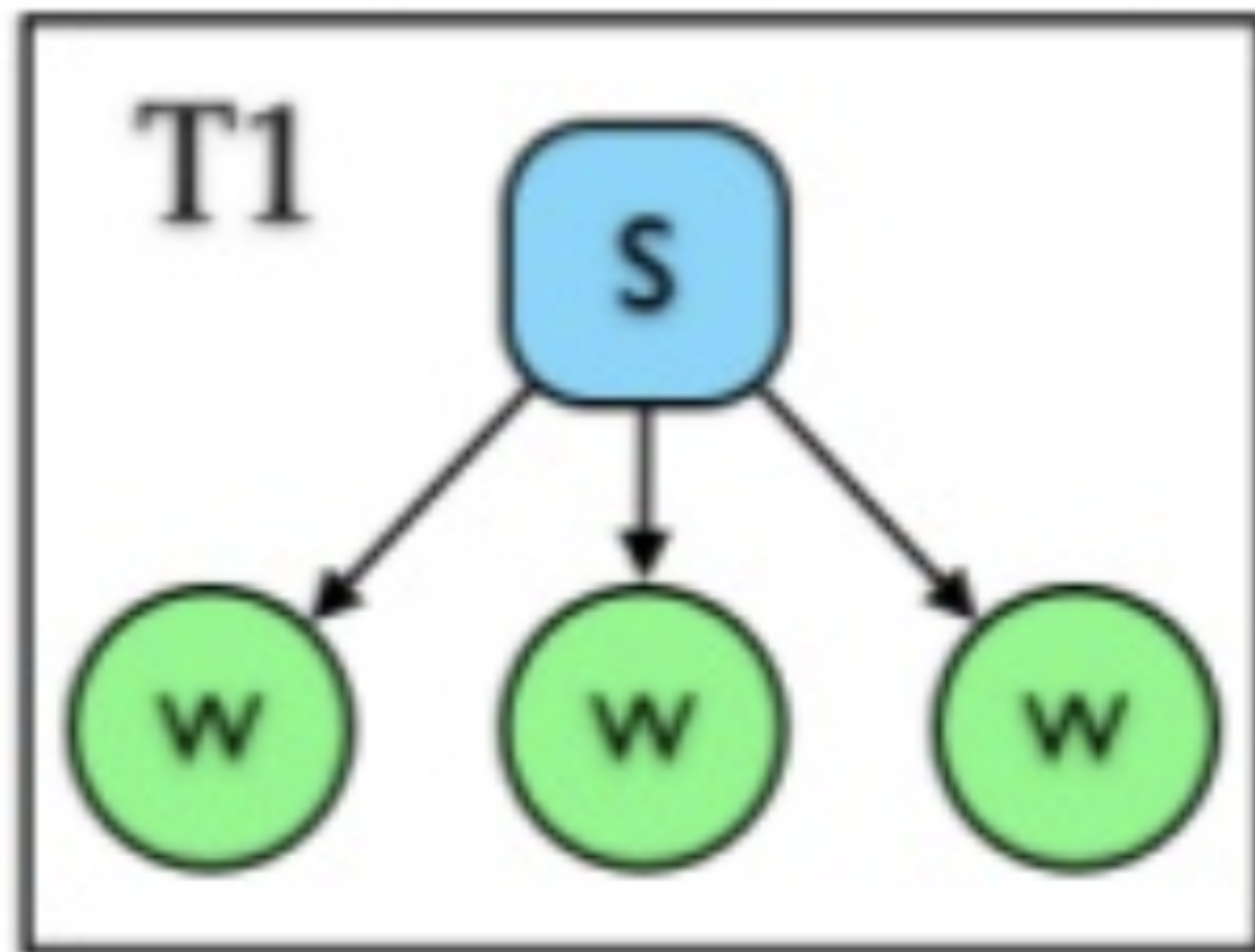
```
Interactive Elixir (1.6.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> {:ok, pid} = CounterServer.start_link
{:ok, #PID<0.129.0>}
iex(2)> CounterServer.increment(pid)
:ok
iex(3)> CounterServer.increment(pid)
:ok
iex(4)> CounterServer.get_value(pid)
2
```

OTP: Supervisor

- Special kind of process
- 1 job: monitor / restart failing processes
- Many possible restart strategies
- Can be used to monitor other supervisors
-> hierarchical structure, fault isolation
- Key to robustness



OTP: Supervisor (one for all strategy)



Phoenix



Phoenix Framework

“Productive. Reliable. Fast.”

Phoenix - concepts

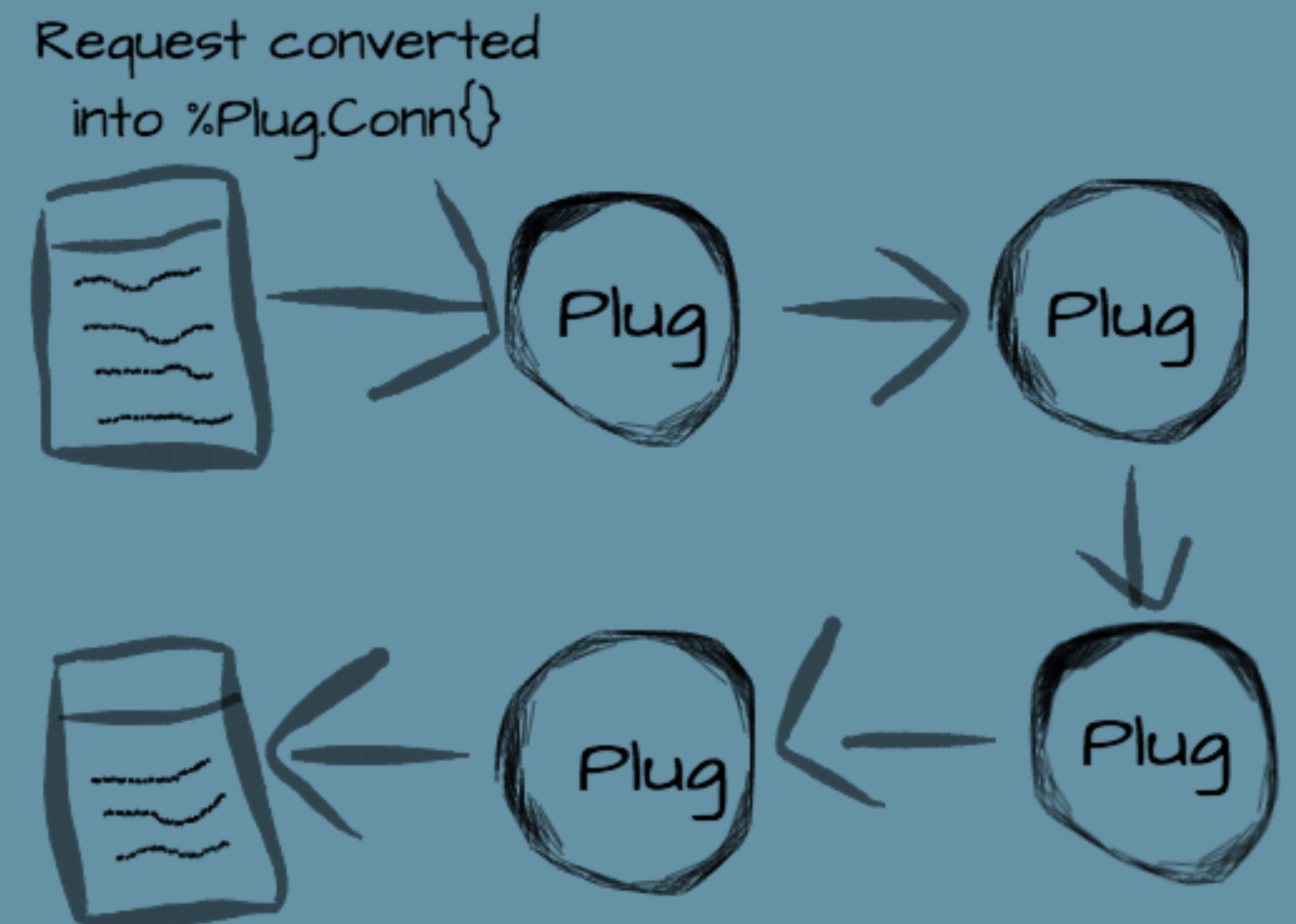
- MVC web framework
- Router DSL
- Built on top of Cowboy / Plug
- Ecto (SQL DSL)

Plug

<https://github.com/elixir-plug/plug>

Plug is:

1. Specification for composable modules between web applications.
2. Each step returns updated Plug.Conn structure
3. Connection adapters for different webserver in the Erlang VM



Plug

```
1 defmodule Api.Authentication do
2   import Plug.Conn
3
4   def init(opts), do: opts
5
6   def call(conn, _opts) do
7     if authenticated?(conn) do
8       conn
9     else
10      conn
11      |> send_resp(401, "")
12      |> halt
13    end
14  end
15
16  def authenticated?(conn) do
17    # Implement authentication logic here,
18    # e.g. check auth header, session creds, etc.
19    false
20  end
21 end
```

- Middleware library
- = 1 action in a Phoenix pipeline
- Used for auth, request validation, logging, ...

Plug

```
1 defmodule HelloPhxWeb.Endpoint do
2   use Phoenix.Endpoint, otp_app: :hello_phx
3
4   plug Plug.Static,
5     at: "/", from: :hello_phx, gzip: false,
6     only: ~w(css fonts images js favicon.ico robots.txt)
7
8   plug Plug.Logger
9
10  plug Plug.Parsers,
11    parsers: [:urlencoded, :multipart, :json],
12    pass: ["*/*"],
13    json_decoder: Poison
14
15  plug Plug.MethodOverride
16  plug Plug.Head
17
18  plug Plug.Session,
19    store: :cookie,
20    key: "_hello_phx_key",
21    signing_salt: "podTC9Bq"
22
23  plug HelloPhxWeb.Router
24
```

- Composable
- Plug pipeline
- Makes Phoenix very explicit

Phoenix Router

```
1 defmodule HelloPhxWeb.Router do
2   use HelloPhxWeb, :router
3
4   pipeline :browser do
5     plug :accepts, ["html"]
6     plug :fetch_session
7     plug :fetch_flash
8     plug :protect_from_forgery
9     plug :put_secure_browser_headers
10  end
11
12  scope "/", HelloPhxWeb do
13    pipe_through :browser # Use the default browser stack
14
15    get "/todos", TodoController, :index
16    get "/todos/:id", TodoController, :show
17    post "/todos", TodoController, :create
18    get "/todos/:id/edit", TodoController, :edit
19    put "/todos/:id", TodoController, :update
20    delete "/todos/:id", TodoController, :delete
21  end
22 end
```

- Easily chain actions for specific endpoints (get, post, ...)
- ``pipeline``: Pipeline macro
- ``pipe_through``: denotes specific pipeline
- Plug specification is imported by ``use HelloPhxWeb, :router``

Ecto

- Declarative DSL for database queries (SQL)
- Inspired by LINQ
- Multiple backends (postgresql, MariaDB, ...)
- Composable queries

```
# Imports only from/2 of Ecto.Query
import Ecto.Query, only: [from: 2]

# Create a query
query = from u in "users",
        where: u.age > 18,
        select: u.name

# Send the query to the repository
Repo.all(query)
```

Elixir & Phoenix at Kabisa

- Fast: sub-millisecond response times
- Minimal mutable state -> less complex systems
- Highly scalable web apps
- Declarative, easy to read code (after getting used to functional thinking)
- Composable
- Con (sometimes): library support / coverage



Elixir/Phoenix is part of Philips Hue Labs (backend)

[FAQ](#)[Contact](#)

... and also part of Wuunder backend



Wuunder

Hans Kenis
Bubblewrap

PRINT LABEL
Print your shipping label!

1

Receiving

Sending

THURSDAY 12/04

MONDAY 09/04

Hans Kenis of **kabisa**
 laptop

MONDAY 26/03

TUESDAY 06/02

Hans Kenis
 Bubblewrap

TUESDAY 12/12

Hans Kenis of **kabisa**
 laptop

FRIDAY 27/10

Send a shipment

hans@kabisa.nl
+32484158495

OK

What is the content of the shipment?

Bubblewrap

Take a picture of what you're sending to Hans Kenis. In doing this you accept our [terms of use](#) for photos.

Skip

Send a personal message to Hans Kenis. This will only be sent when the shipment booking is completed.

Have fun!

What do you want to send?

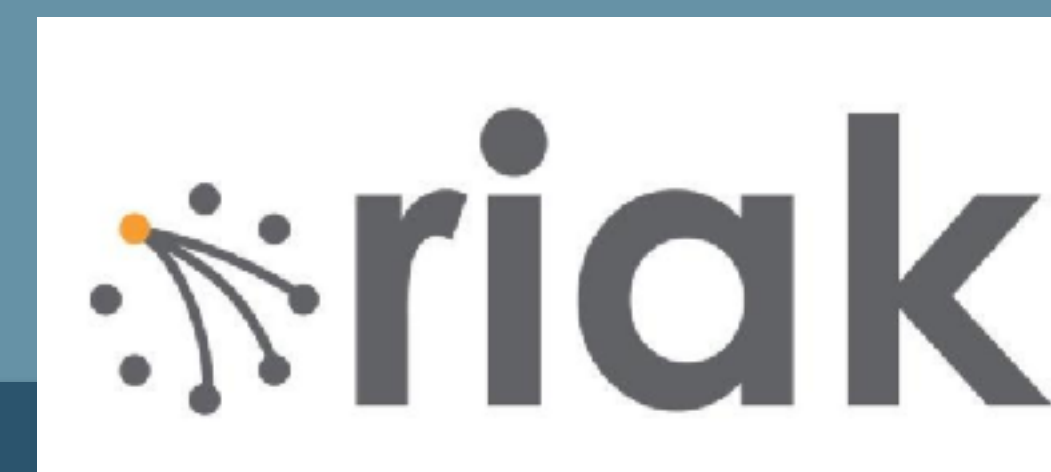
Parcel within the Netherlands
(Max 20kg and max 80 x 50 x 35 cm)

Who can we collect the shipment from?

Hans Kenis
Marconilaan, Weert

Search in address book

Erlang & Elixir in industry



And many more..!

Demo

- Supervisor / observer example
- Processes example (Pingpong example + Visualixir)

Reading material

- <https://elixir-lang.org/>
- <http://phoenixframework.org/>
- <https://kabisa.nl/>
- <https://theguild.nl>
- <https://www.meetup.com/nl-NL/Weert-Software-Development-Meetup/>



Workshop dinsdag 29 mei

Running applications on AWS setup with Docker and Terraform

Q & A



Luc Tielen
@luctielen



Hans Kenis
@HansKenis