OHA!

ОНЕЈ!

Implementing A Worker Pool Application in 4 acts

or: How I finally grokked OTP

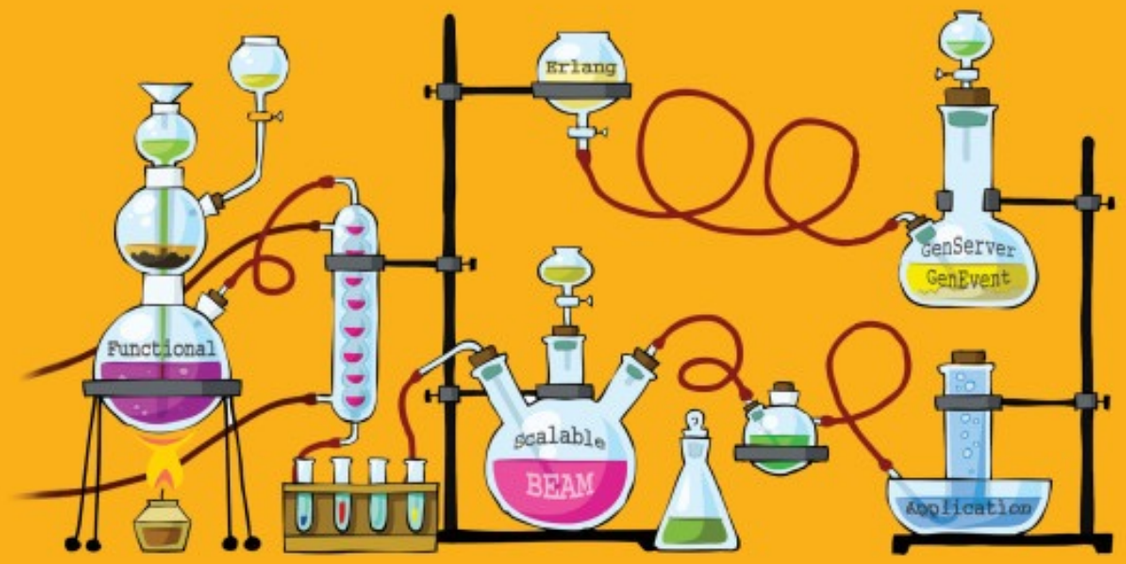
9th June 2017

Benjamin Tan Wei Hao
@bentanweihao

Erlang User Conference 2017

Erlang USER CONFERENCE

THE LITTLE Elixir & OTP GUIDEBOOK



Benjamin Tan Wei Hao

MANNING

13 hours!

WHAT IS A WORKER POOL?

WHY BUILD A WORKER POOL?

Poolboy

< 400 lines!

is a **lightweight**, **generic** pooling library for Erlang with a focus on **simplicity**, **performance**, and **rock-solid** disaster recovery.

Terminology

① Check-out

② Check-in

Terminology

① Check-out

② Check-in

Creating a Process:

```
iex(1)> {:ok, pid} = ChuckFetcher.start_link(:ok)
iex(2)> ChuckFetcher.fetch(pid)
"Chuck Norris can instantiate an abstract class.”
```

Creating a Process:

```
iex(1)> {:ok, pid} = ChuckFetcher.start_link(:ok)
iex(2)> ChuckFetcher.fetch(pid)
"Chuck Norris can instantiate an abstract class.”
```

Creating a Process:

```
iex(1)> {:ok, pid} = ChuckFetcher.start_link(:ok)
iex(2)> ChuckFetcher.fetch(pid)
"Chuck Norris can instantiate an abstract class.”
```

Checking Out & In a Process:

```
iex(1)> pid = Pooly.checkout("ChuckNorris")
#PID<0.180.0>

iex(2)> ChuckFetcher.fetch(pid)
"Chuck Norris can unit test entire applications with a single assert."

iex(3)> Pooly.checkin("ChuckNorris", pid)
:ok
```

Creating a Process:

```
iex(1)> {:ok, pid} = ChuckFetcher.start_link(:ok)
iex(2)> ChuckFetcher.fetch(pid)
"Chuck Norris can instantiate an abstract class.”
```

Checking Out & In a Process:

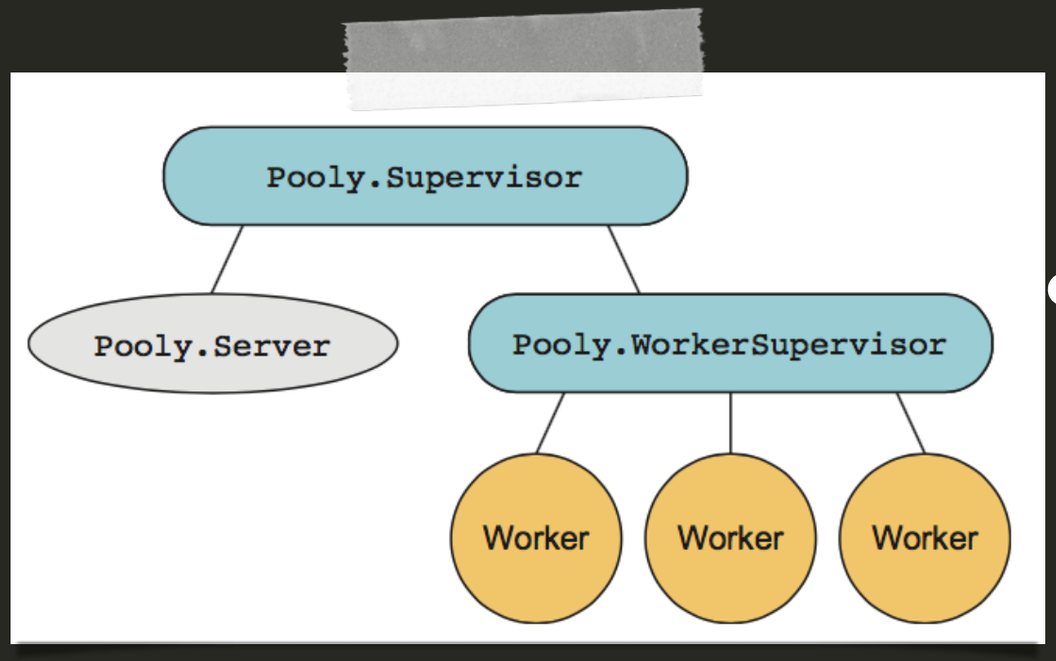
```
iex(1)> pid = Pooly.checkout("ChuckNorris")
#PID<0.180.0>

iex(2)> ChuckFetcher.fetch(pid)
"Chuck Norris can unit test entire applications with a single assert."

iex(3)> Pooly.checkin("ChuckNorris", pid)
:ok
```

Creating a Process:

```
iex(1)> {:ok, pid} = ChuckFetcher.start_link(:ok)
iex(2)> ChuckFetcher.fetch(pid)
"Chuck Norris can instantiate an abstract class.”
```

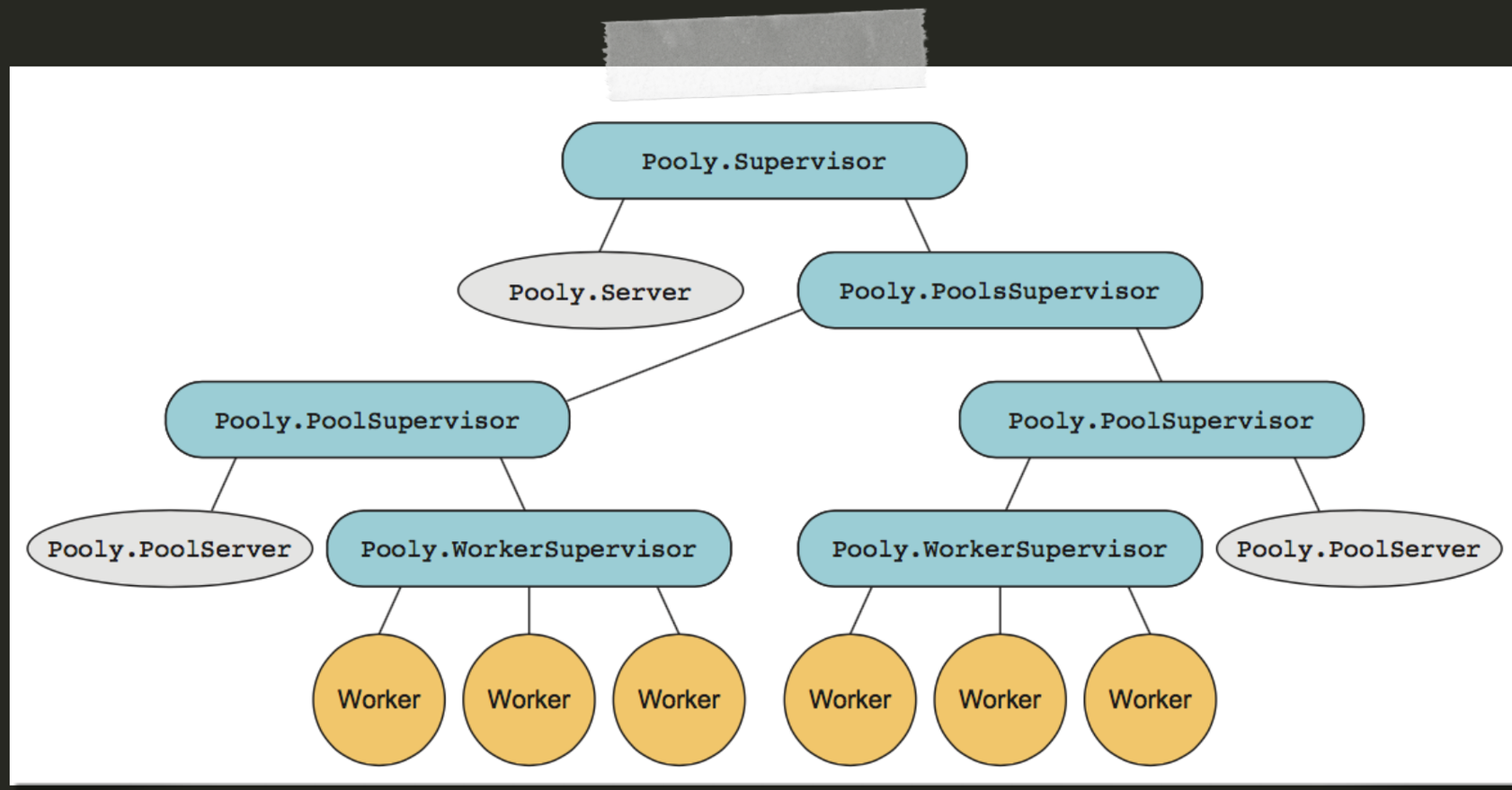
Checking Out & In a Process:

```
iex(1)> pid = Pooly.checkout("ChuckNorris")
#PID<0.180.0>

iex(2)> ChuckFetcher.fetch(pid)
"Chuck Norris can unit test entire applications with a single assert."

iex(3)> Pooly.checkin("ChuckNorris", pid)
:ok
```



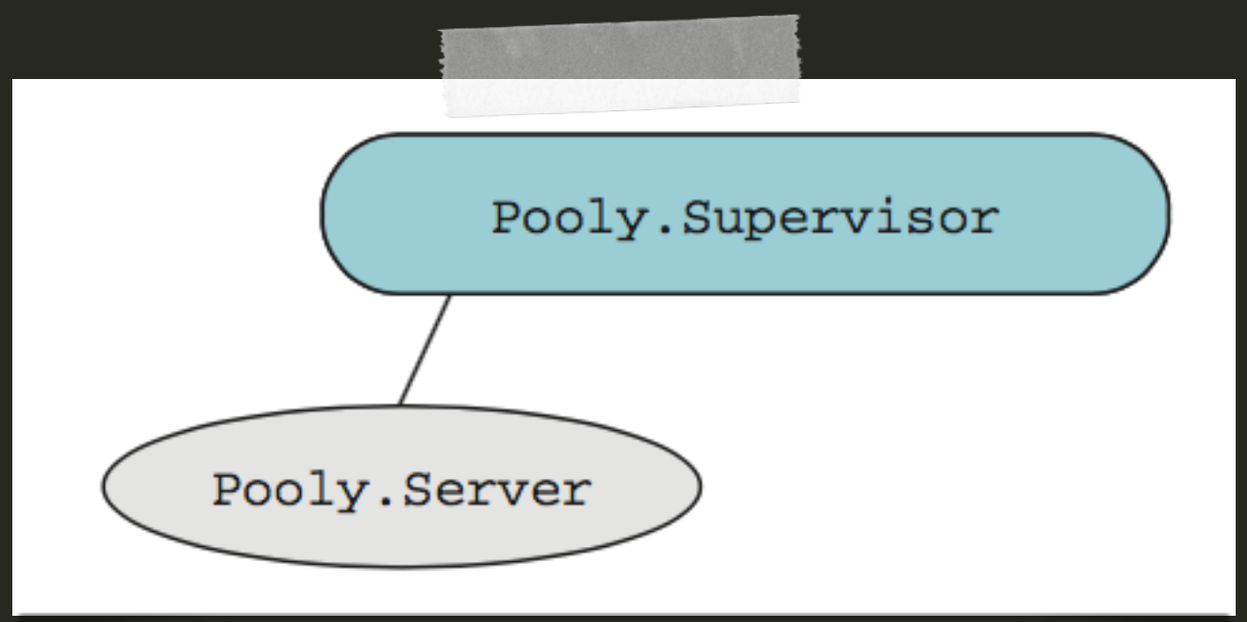
VERSION 1 & 2

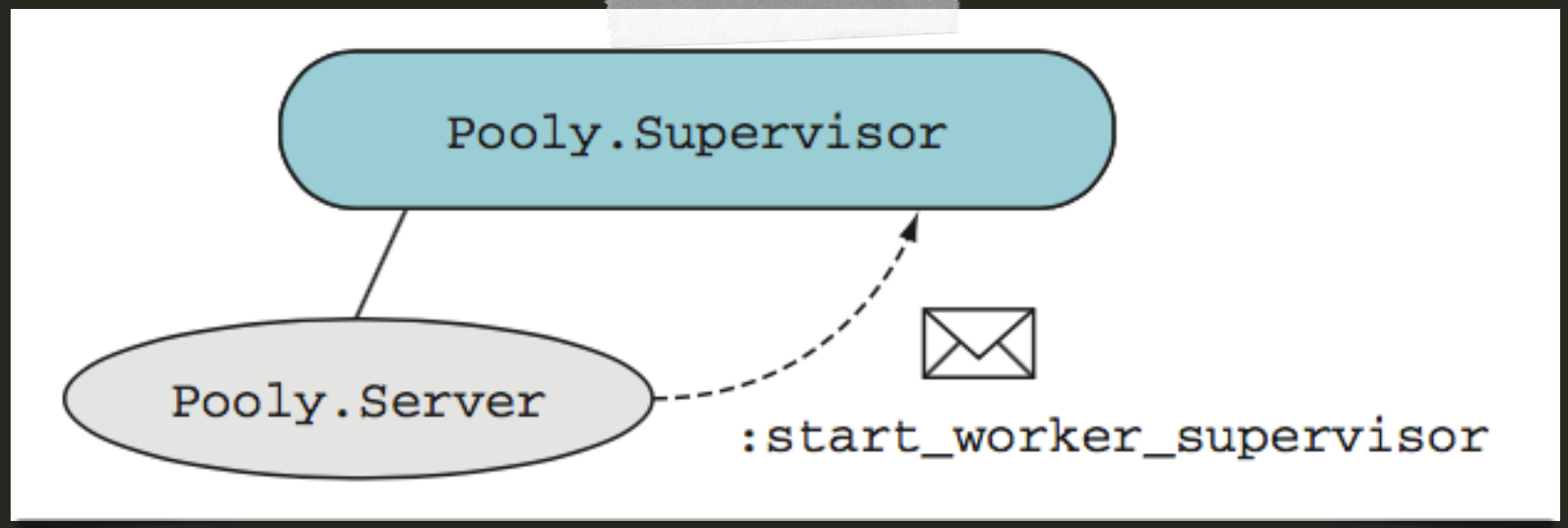
VERSION 3 & 4

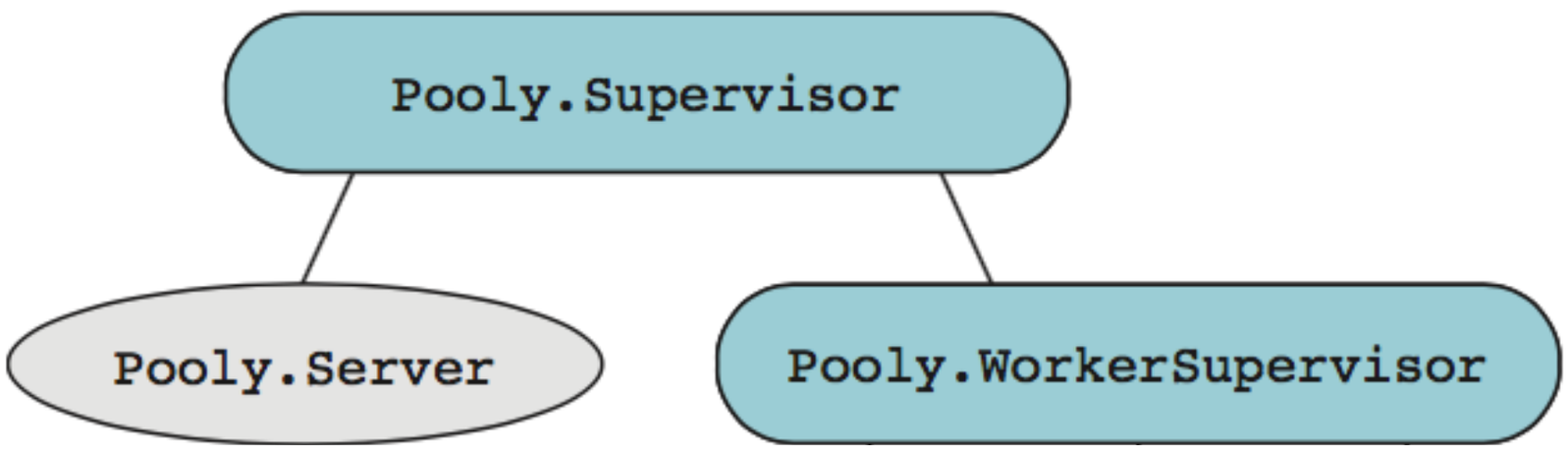


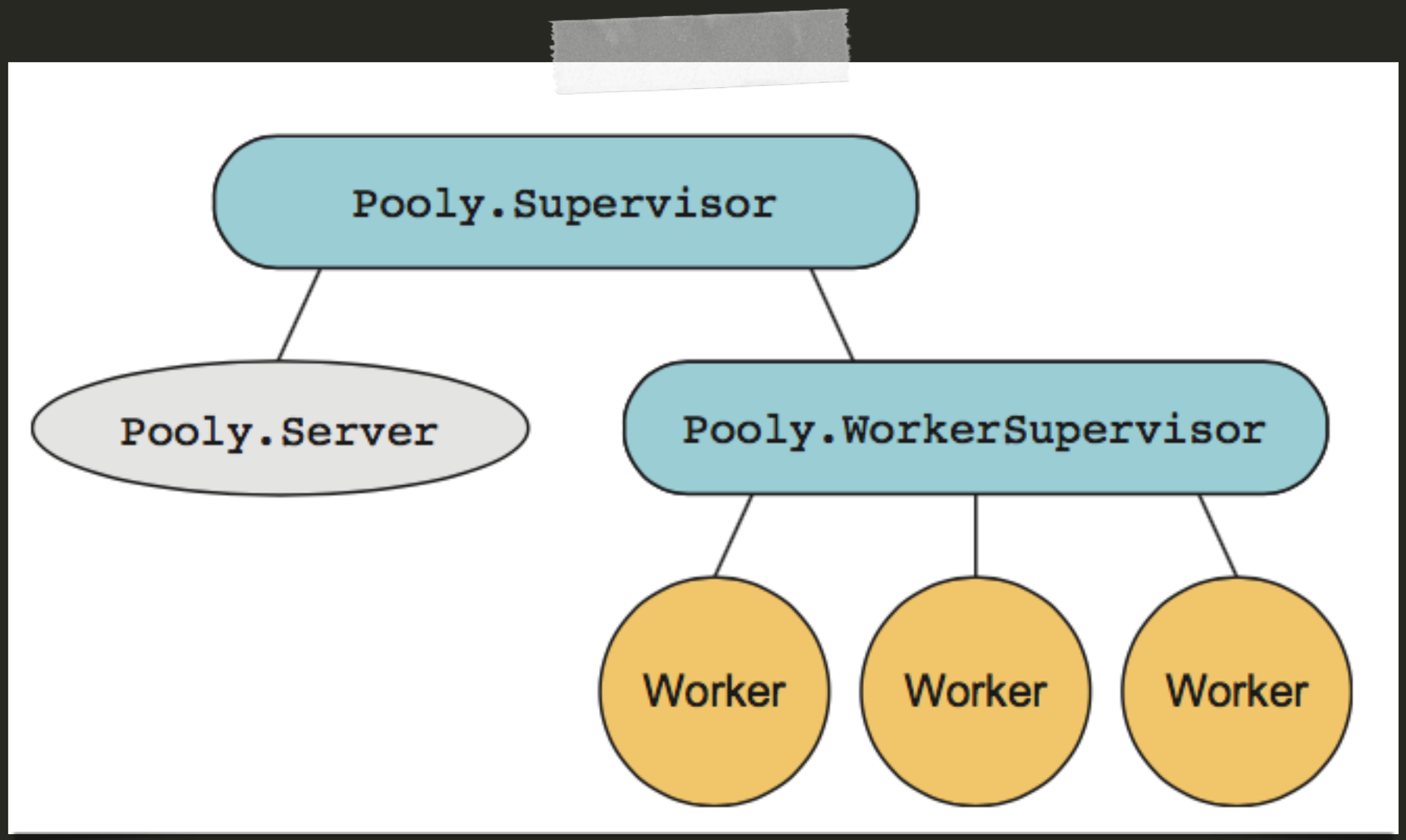
VERSION 1

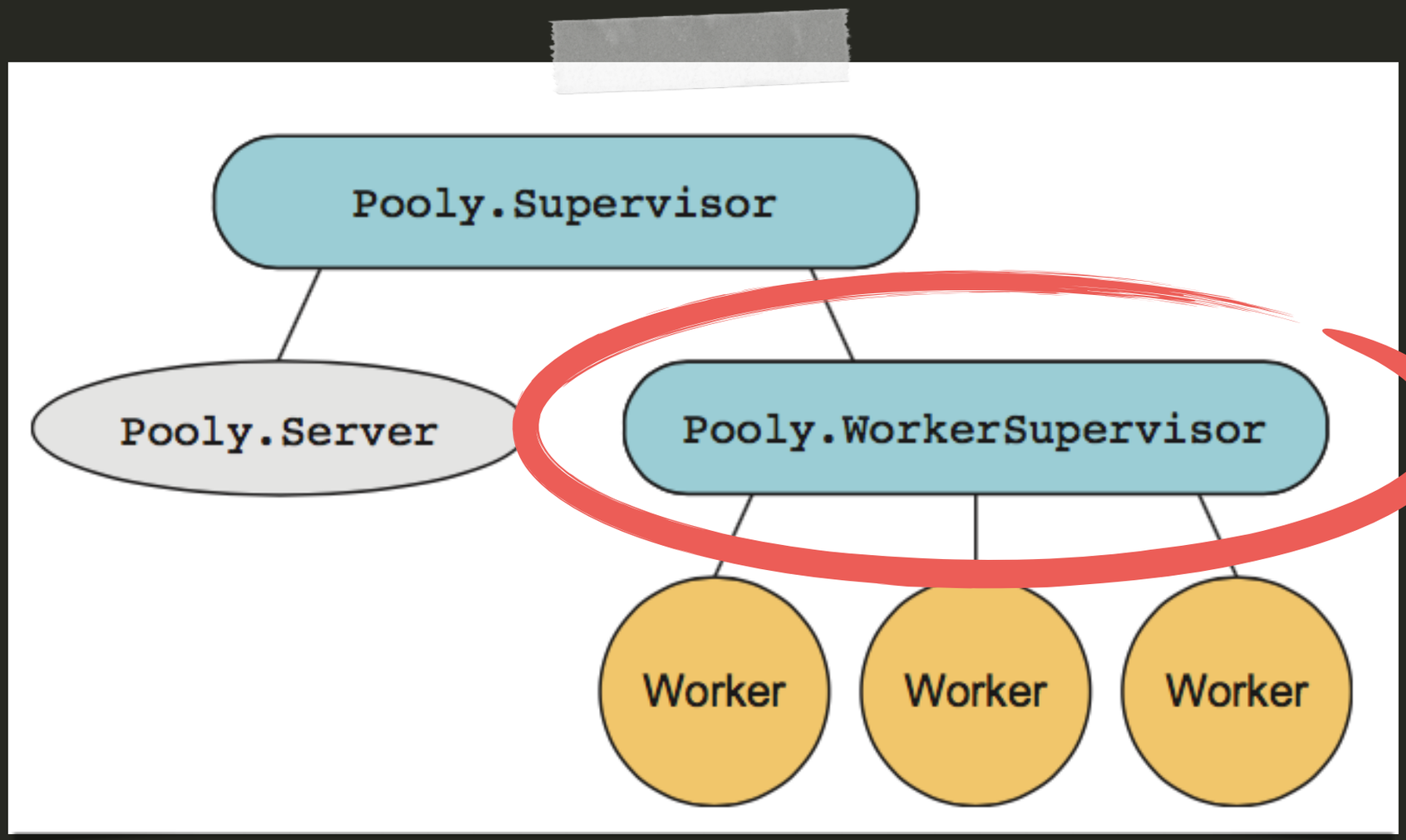
TYPE OF POOL	Single	Multiple
CREATION OF WORKERS	Fixed	Dynamic
CONSUMER RECOVERY	No	Yes
WORKER RECOVERY	No	Yes
QUEUEING FOR BUSY WORKERS	No	Yes

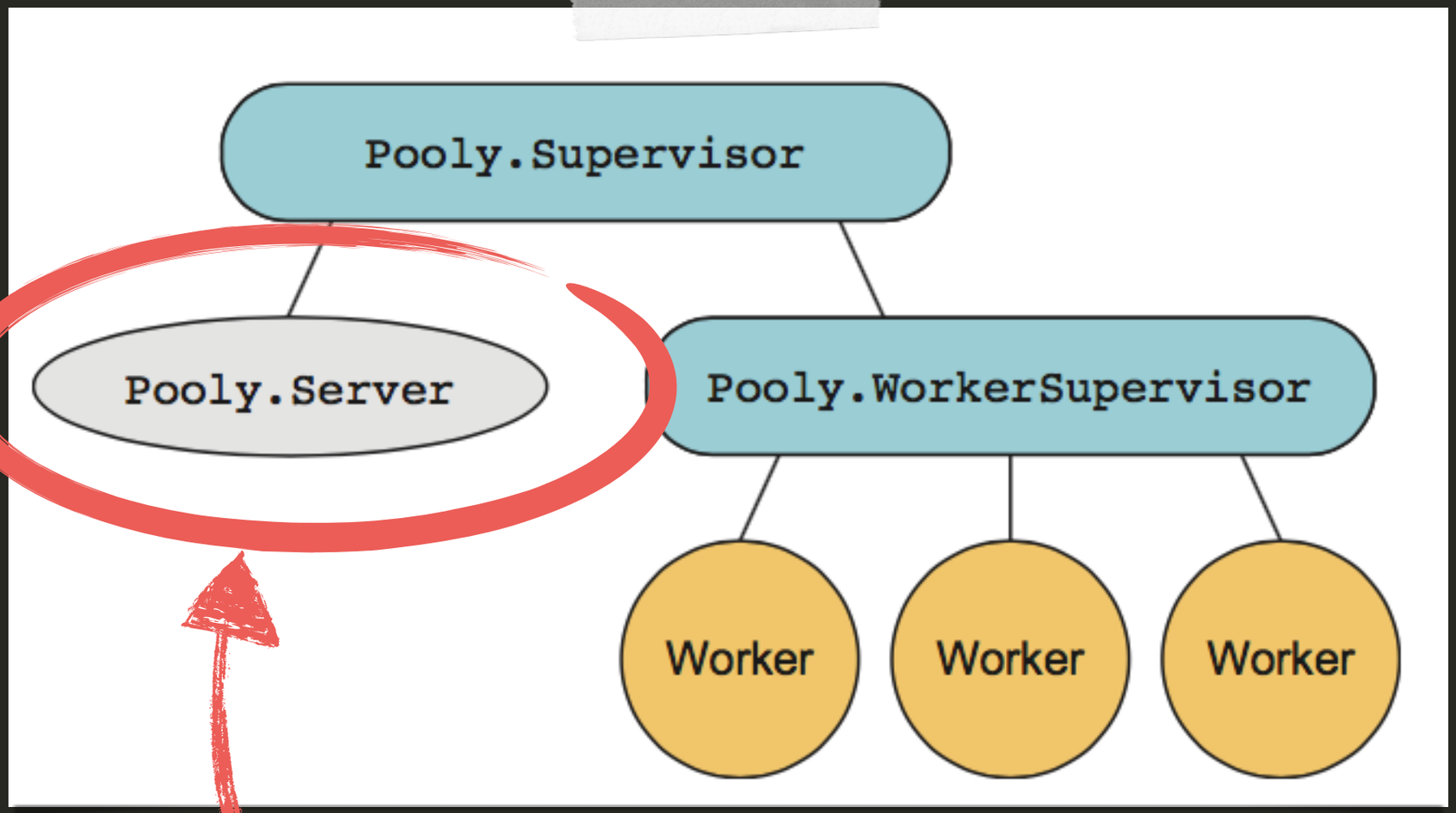

Pooly.Supervisor
Pooly.Server


Pooly.Supervisor
Pooly.Server
:start_worker_supervisor


Pooly.Supervisor
Pooly.Server
Pooly.WorkerSupervisor


Pooly.Supervisor
Pooly.Server
Pooly.WorkerSupervisor
Worker
Worker
Worker

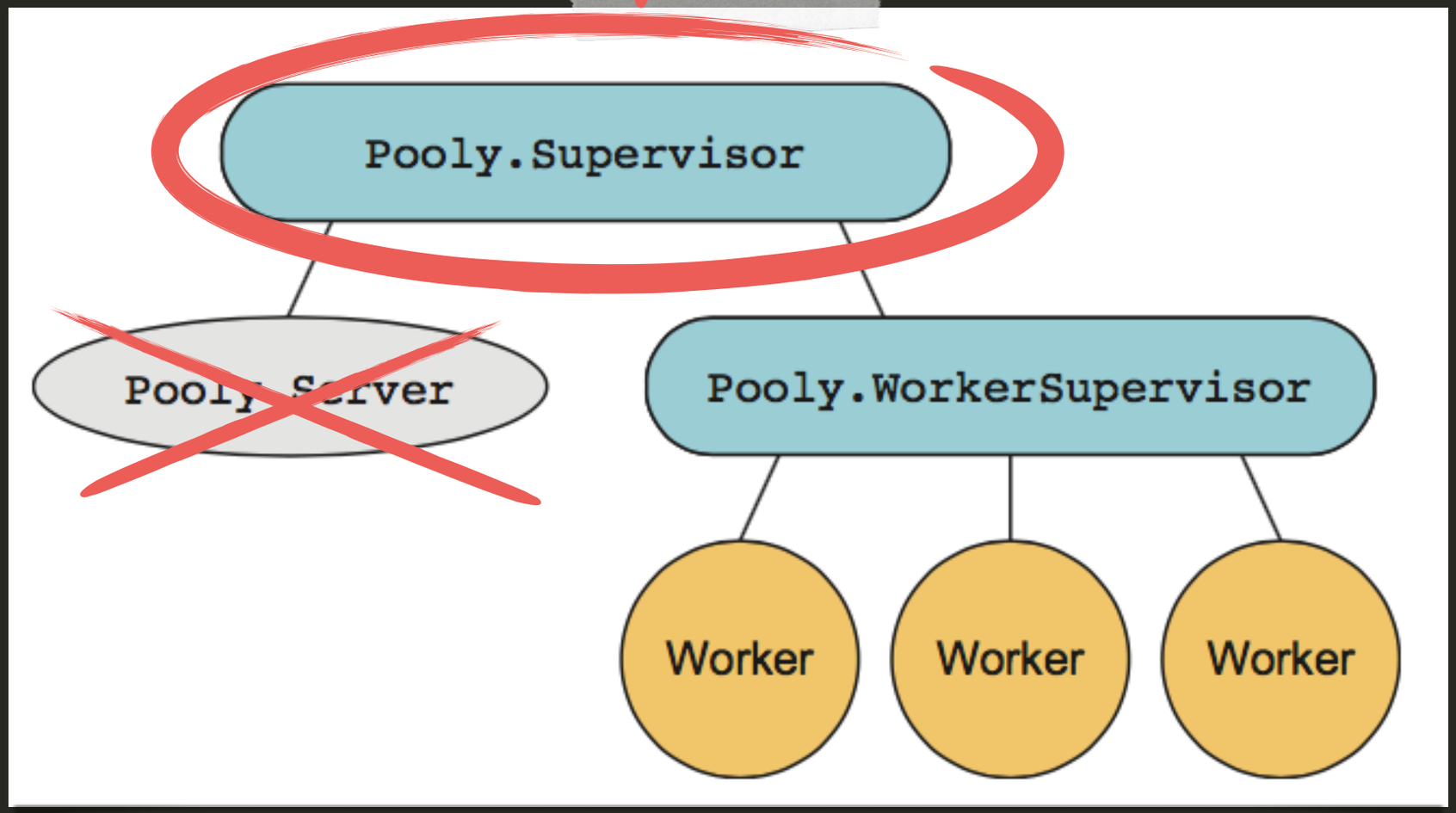

Pooly.Supervisor
Pooly.Server
Pooly.WorkerSupervisor
Worker
Worker
Worker



REALLY NEEDED?

```
-module(supervisor).

-behaviour(gen_server).

-export([start_link/2, start_link/3,
    start_child/2, restart_child/2,
    delete_child/2, terminate_child/2,
    which_children/1, count_children/1,
    check_childspecs/1, get_childspec/2]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
    terminate/2, code_change/3, format_status/2]).
-export([try_again_restart/2]).
```

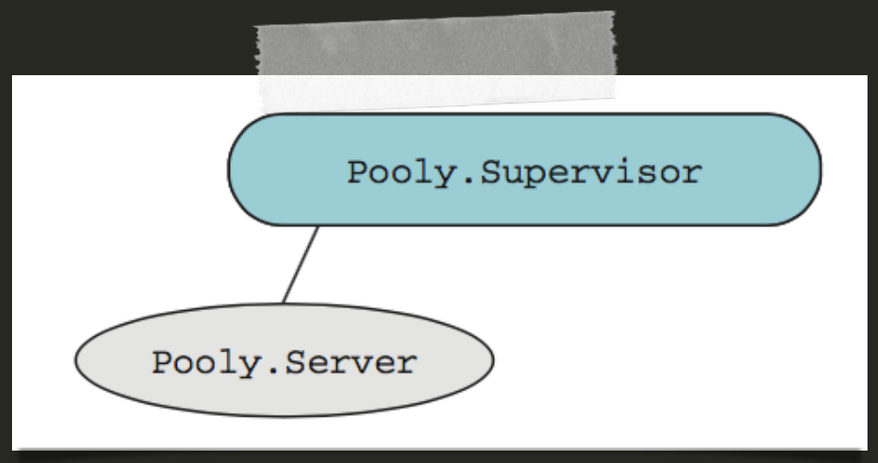
LOGIC GOES HERE?

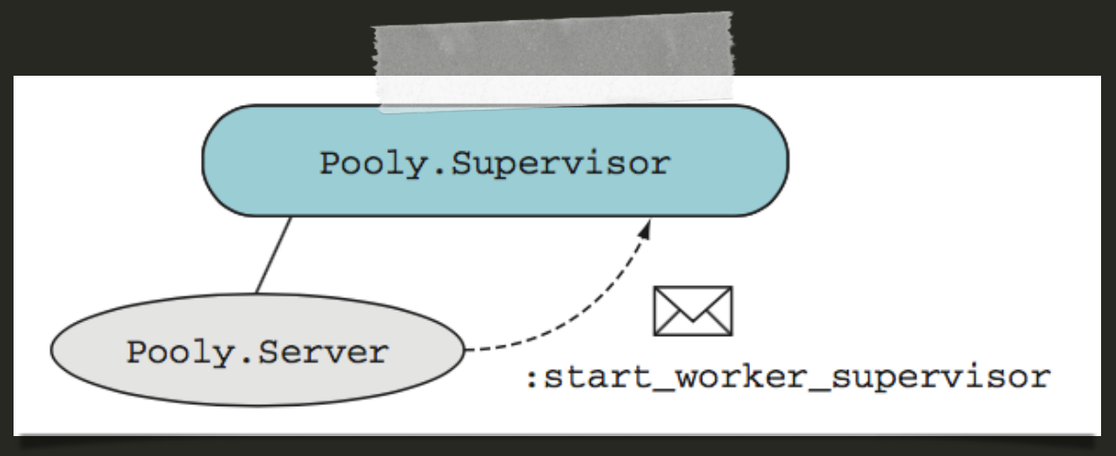


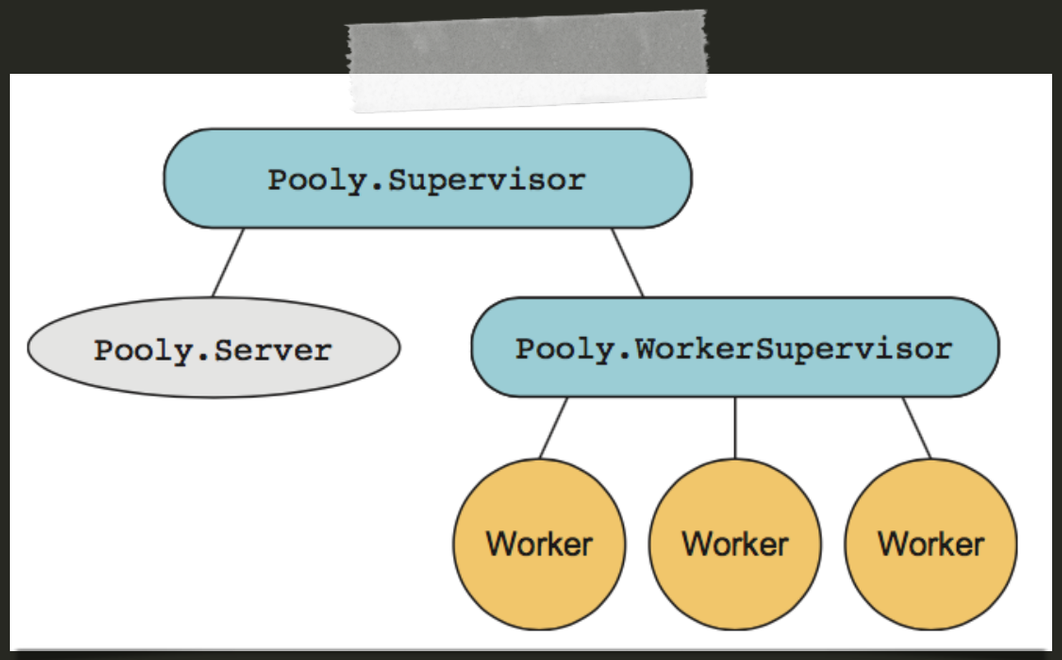
IF DEBUGGING IS THE PROCESS
OF REMOVING BUGS FROM THE CODE

THE PROGRAMMING MUST BE THE
PROCESS OF PUTTING THEM IN

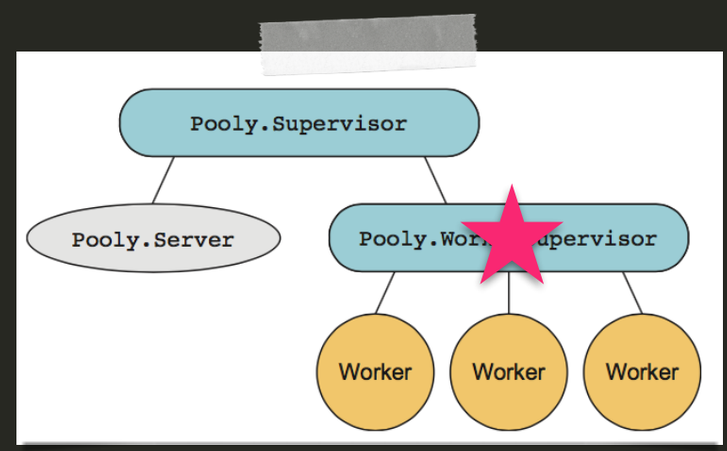
WHAT GOES INTO THE SERVER STATE?


Pooly.Supervisor
Pooly.Server

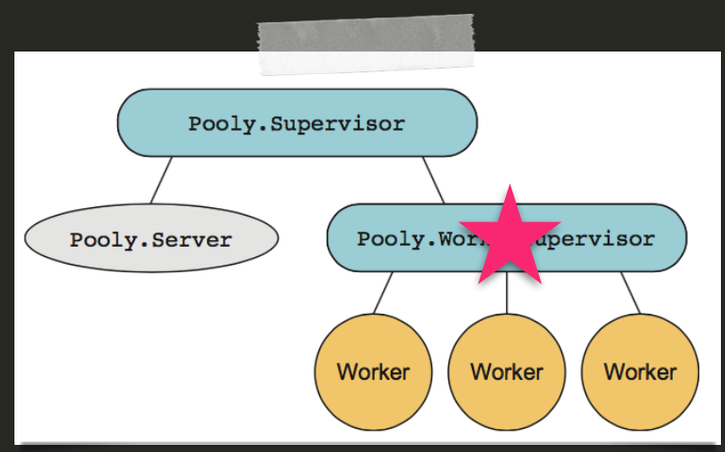

Pooly.Supervisor
Pooly.Server
:start_worker_supervisor


Pooly.Supervisor
Pooly.Server
Pooly.WorkerSupervisor
Worker
Worker
Worker

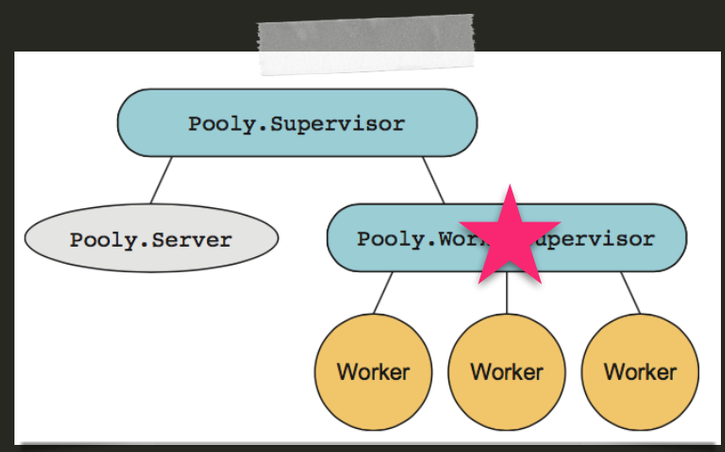
```elixir
defmodule Pooly.WorkerSupervisor do
  use Supervisor

  def start_link({_,_,_} = mfa) do
    Supervisor.start_link(__MODULE__, mfa)
  end

  def init({m,f,a}) do
    worker_opts = [restart:  :permanent,
                   shutdown: 5000,
                   function: f]

    children = [worker(m, a, worker_opts)]
    opts     = [strategy:     :simple_one_for_one,
                max_restarts: 5,
                max_seconds:  5]

    supervise(children, opts)
  end
end
```



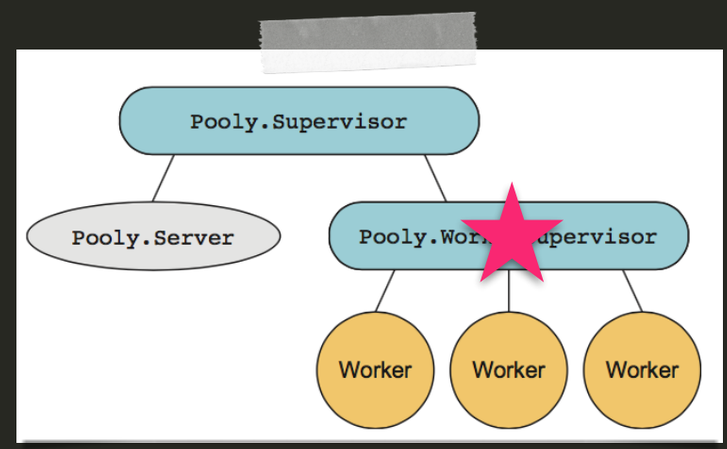
```elixir
defmodule Pooly.WorkerSupervisor do
  use Supervisor

  def start_link({_,_,_} = mfa) do
    Supervisor.start_link(__MODULE__, mfa)
  end

  def init({m,f,a}) do
    worker_opts = [restart:   :permanent,
                   shutdown: 5000,
                   function: f]

    children = [worker(m, a, worker_opts)]
    opts     = [strategy:      :simple_one_for_one,
                max_restarts: 5,
                max_seconds:  5]

    supervise(children, opts)
  end
end
```



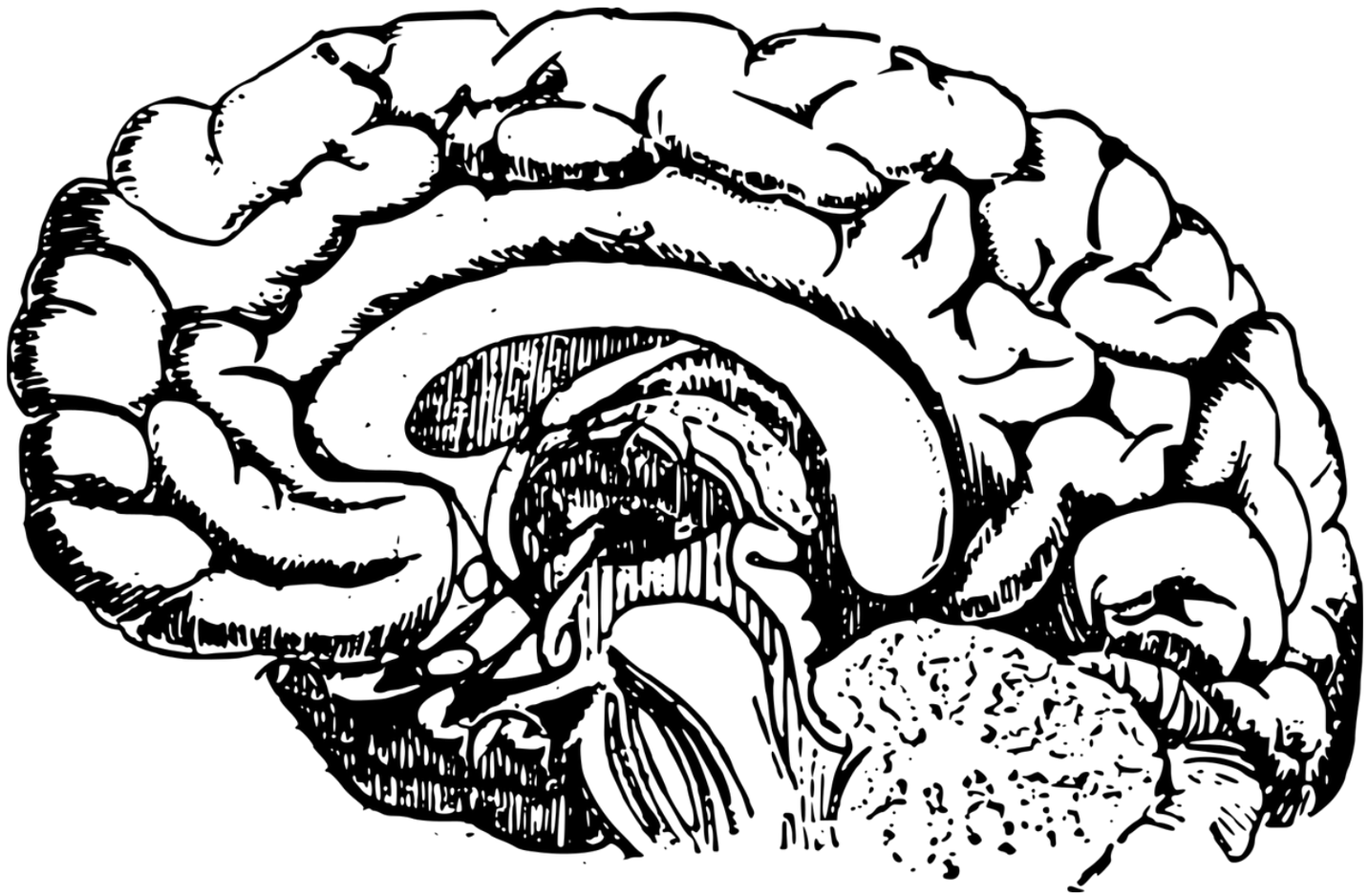
```elixir
defmodule Pooly.WorkerSupervisor do
  use Supervisor

  def start_link({_,_,_} = mfa) do
    Supervisor.start_link(__MODULE__, mfa)
  end

  def init({m,f,a}) do
    worker_opts = [restart:  :permanent,
                   shutdown: 5000,
                   function: f]

    children = [worker(m, a, worker_opts)]
    opts     = [strategy:     :simple_one_for_one,
                max_restarts: 5,
                max_seconds:  5]

    supervise(children, opts)
  end
end
```



```elixir
defmodule Pooly.WorkerSupervisor do
  use Supervisor

  def start_link({_,_,_} = mfa) do
    Supervisor.start_link(__MODULE__, mfa)
  end

  def init({m,f,a}) do
    worker_opts = [restart:  :permanent,
                   shutdown: 5000,
                   function: f]

    children = [worker(m, a, worker_opts)]
    opts     = [strategy:     :simple_one_for_one,
                max_restarts: 5,
                max_seconds:  5]

    supervise(children, opts)
  end
end
```

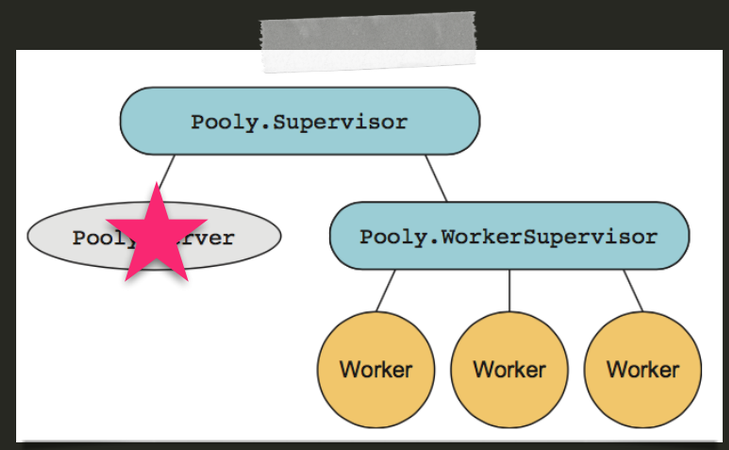


The Pool Server:

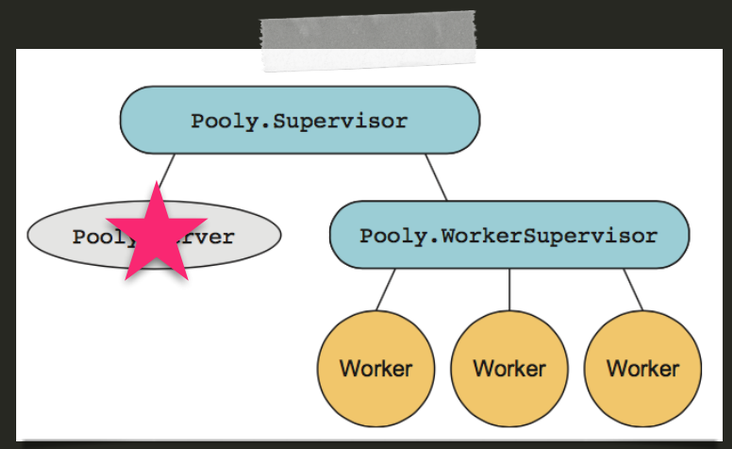


OF THE OPERATION

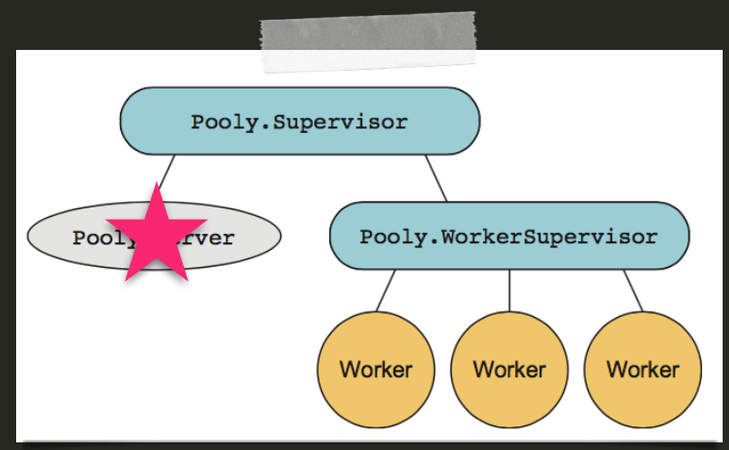
```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  defmodule State do
    defstruct sup: nil, size: nil, mfa: nil
  end

  def start_link(sup, pool_config) do
    GenServer.start_link(__MODULE__, [sup, pool_config],
                         name: __MODULE__)
  end

  def init([sup, pool_config]) when is_pid(sup) do
    init(pool_config, %{sup: sup})
  end
  def init([{:mfa, mfa}|rest], state) do: init(rest, %{state | mfa: mfa})
  def init([{:size, s}|rest], state), do: init(rest, %{state | size: s})
  def init([_|rest], state), do: init(rest, state)
  def init([], state) do
    send(self, :start_worker_supervisor)
    {:ok, state}
  end
end
```



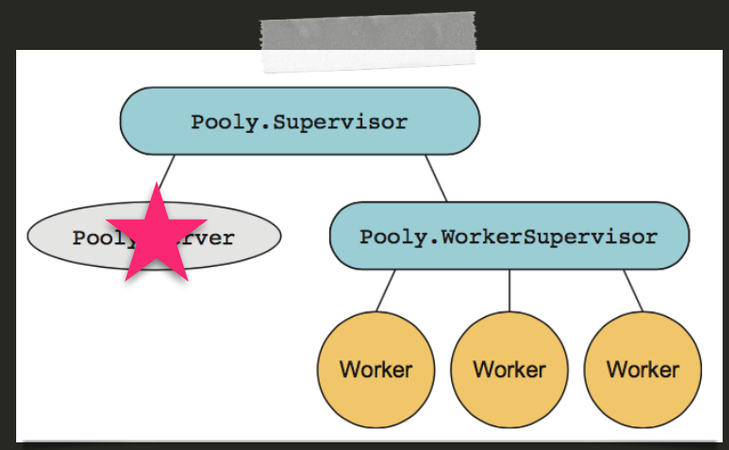
```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  defmodule State do
    defstruct sup: nil, size: nil, mfa: nil
  end

  def start_link(sup, pool_config) do
    GenServer.start_link(__MODULE__, [sup, pool_config],
                         name: __MODULE__)
  end

  def init([sup, pool_config]) when is_pid(sup) do
    init(pool_config, %{sup: sup})
  end
  def init([{:mfa, mfa}|rest], state) do: init(rest, %{state | mfa: mfa})
  def init([{:size, s}|rest], state), do: init(rest, %{state | size: s})
  def init([_|rest], state), do: init(rest, state)
  def init([], state) do
    send(self, :start_worker_supervisor)
    {:ok, state}
  end
end
```



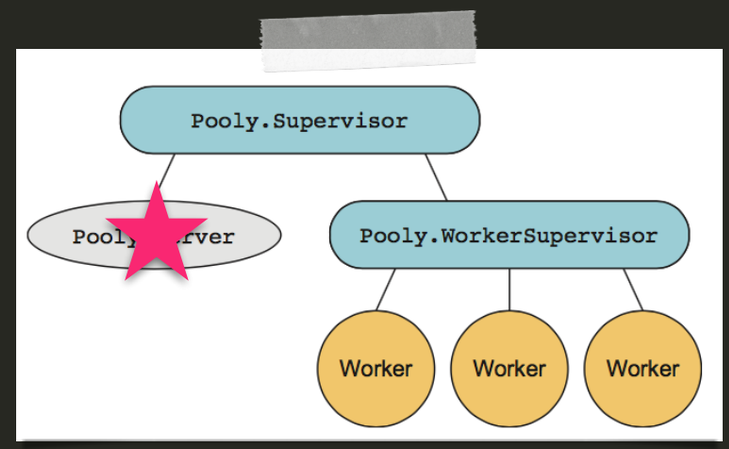
```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  defmodule State do
    defstruct sup: nil, size: nil, mfa: nil
  end

  def start_link(sup, pool_config) do
    GenServer.start_link(__MODULE__, [sup, pool_config],
                         name: __MODULE__)
  end

  def init([sup, pool_config]) when is_pid(sup) do
    init(pool_config, %{sup: sup})
  end

  def init([{:mfa, mfa}|rest], state) do: init(rest, %{state | mfa: mfa})
  def init([{:size, s}|rest], state), do: init(rest, %{state | size: s})
  def init([_|rest], state), do: init(rest, state)
  def init([], state) do
    send(self, :start_worker_supervisor)
    {:ok, state}
  end
end
```



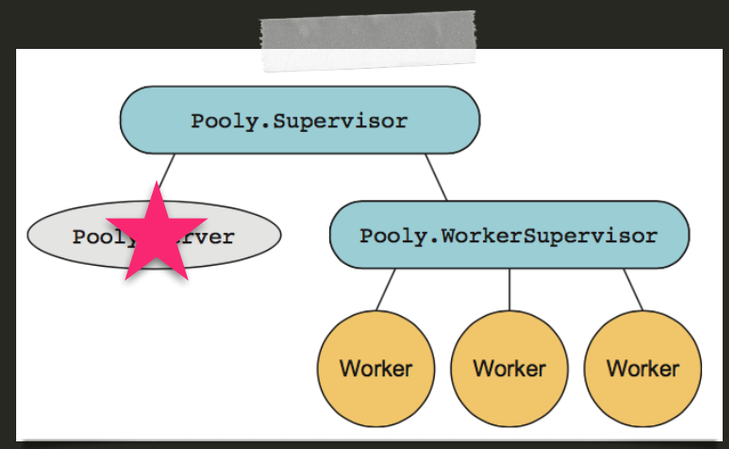
```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  defmodule State do
    defstruct sup: nil, size: nil, mfa: nil
  end

  def start_link(sup, pool_config) do
    GenServer.start_link(__MODULE__, [sup, pool_config],
                         name: __MODULE__)
  end

  def init([sup, pool_config]) when is_pid(sup) do
    init(pool_config, %{sup: sup})
  end
  def init([{:mfa, mfa}|rest], state) do: init(rest, %{state | mfa: mfa})
  def init([{:size, s}|rest], state), do: init(rest, %{state | size: s})
  def init([_|rest], state), do: init(rest, state)
  def init([], state) do
    send(self, :start_worker_supervisor)
    {:ok, state}
  end
end
```



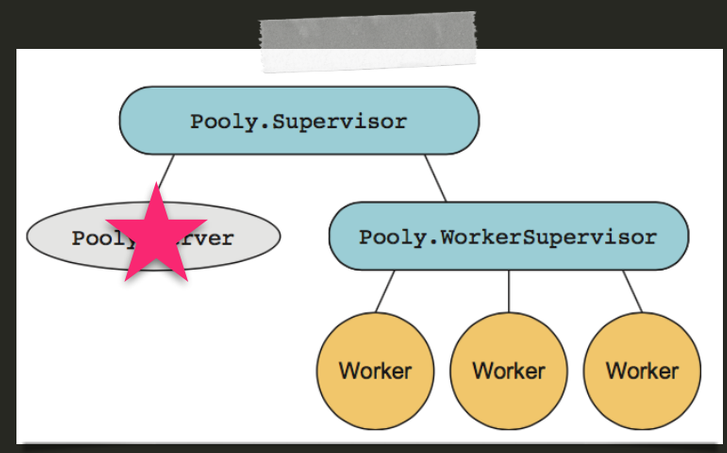
```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  defmodule State do
    defstruct sup: nil, size: nil, mfa: nil
  end

  def start_link(sup, pool_config) do
    GenServer.start_link(__MODULE__, [sup, pool_config],
                         name: __MODULE__)
  end

  def init([sup, pool_config]) when is_pid(sup) do
    init(pool_config, %{sup: sup})
  end
  def init([{:mfa, mfa}|rest], state) do: init(rest, %{state | mfa: mfa})
  def init([{:size, s}|rest], state), do: init(rest, %{state | size: s})
  def init([], state) do
  def init([], state) do
    send(self, :start_worker_supervisor)
    {:ok, state}
  end
end
```



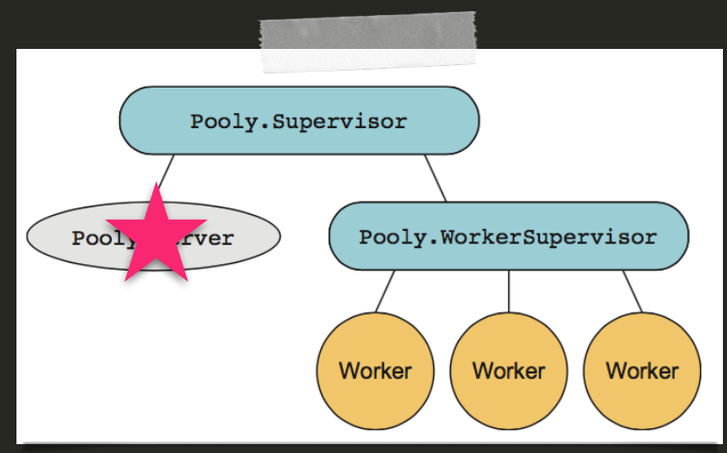
```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  defmodule State do
    defstruct sup: nil, size: nil, mfa: nil
  end

  def start_link(sup, pool_config) do
    GenServer.start_link(__MODULE__, [sup, pool_config],
                         name: __MODULE__)
  end

  def init([sup, pool_config]) when is_pid(sup) do
    init(pool_config, %{sup: sup})
  end
  def init([{:mfa, mfa}|rest], state) do: init(rest, %{state | mfa: mfa})
  def init([{:size, s}|rest], state), do: init(rest, %{state | size: s})
  def init([], state) do te), do: init(rest, state)
    send(self, :start_worker_supervisor)
    {:ok, state} tart_worker_supervisor)
  end:ok, state}
  end
end
```



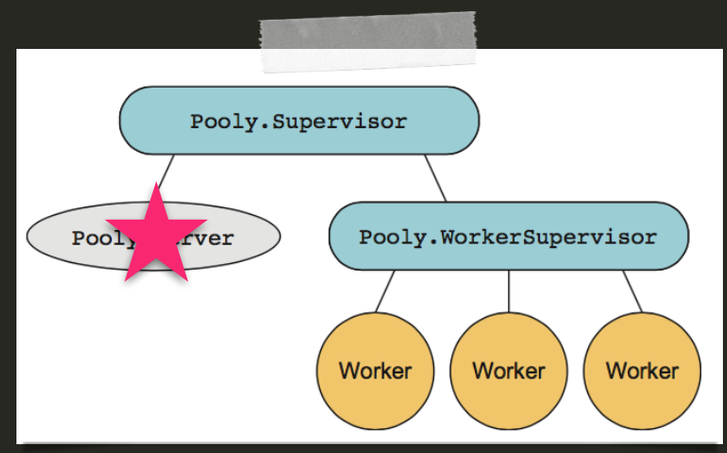
```elixir
defmodule Pooly.Server do
  defstruct State do
    sup: nil, worker_sup: nil, size: nil, workers: nil, mfa: nil
  end

  def handle_info(:start_worker_supervisor, state) do
    %{sup: sup, mfa: mfa, size: size} = state
    {:ok, worker_sup} = Supervisor.start_child(sup, supervisor_spec(mfa))
    workers = prepopulate(size, worker_sup)
    {:noreply, %{state | worker_sup: worker_sup, workers: workers}}
  end

  defp prepopulate(size, sup) when size > 1 do
    1..size |> Enum.map(fn _ -> new_worker(sup) end)
  end
  defp prepopulate(_size, _sup), do: []

  defp new_worker(sup) do
    {:ok, worker} = Supervisor.start_child(sup, [[]])
    worker
  end

  defp supervisor_spec(mfa) do
    supervisor(Pooly.WorkerSupervisor, [mfa],
               [restart: :temporary])
  end
end
```



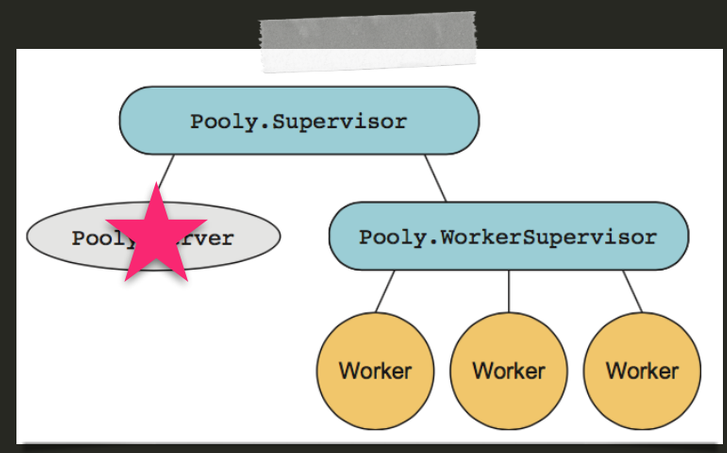
```elixir
defmodule Pooly.Server do
  defstruct State do
    sup: nil, worker_sup: nil, size: nil, workers: nil, mfa: nil
  end

  def handle_info(:start_worker_supervisor, state) do
    %{sup: sup, mfa: mfa, size: size} = state
    {:ok, worker_sup} = Supervisor.start_child(sup, supervisor_spec(mfa))
    workers = prepopulate(size, worker_sup)
    {:noreply, %{state | worker_sup: worker_sup, workers: workers}}
  end

  defp prepopulate(size, sup) when size > 1 do
    1..size |> Enum.map(fn _ -> new_worker(sup) end)
  end
  defp prepopulate(_size, _sup), do: []

  defp new_worker(sup) do
    {:ok, worker} = Supervisor.start_child(sup, [[]])
    worker
  end

  defp supervisor_spec(mfa) do
    supervisor(Pooly.WorkerSupervisor, [mfa],
               [restart: :temporary])
  end
end
```



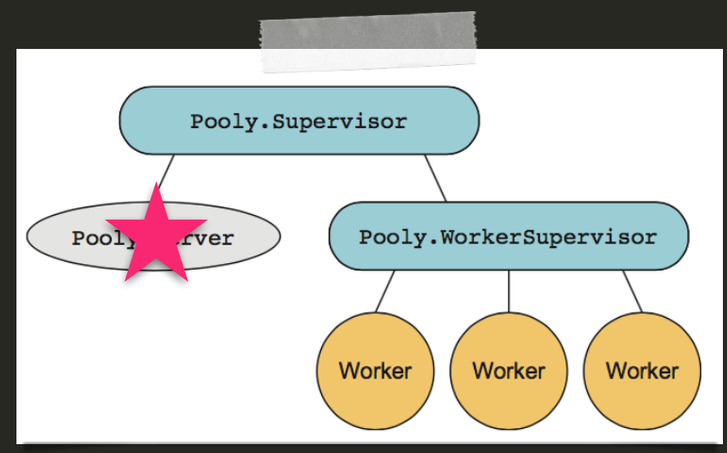
```elixir
defmodule Pooly.Server do
  defstruct State do
    sup: nil, worker_sup: nil, size: nil, workers: nil, mfa: nil
  end

  def handle_info(:start_worker_supervisor, state) do
    %{sup: sup, mfa: mfa, size: size} = state
    {:ok, worker_sup} = Supervisor.start_child(sup, supervisor_spec(mfa))
    workers = prepopulate(size, worker_sup)
    {:noreply, %{state | worker_sup: worker_sup, workers: workers}}
  end

  defp prepopulate(size, sup) when size > 1 do
    1..size |> Enum.map(fn _ -> new_worker(sup) end)
  end
  defp prepopulate(_size, _sup), do: []

  defp new_worker(sup) do
    {:ok, worker} = Supervisor.start_child(sup, [[]])
    worker
  end

  defp supervisor_spec(mfa) do
    supervisor(Pooly.WorkerSupervisor, [mfa],
               [restart: :temporary])
  end
end
```



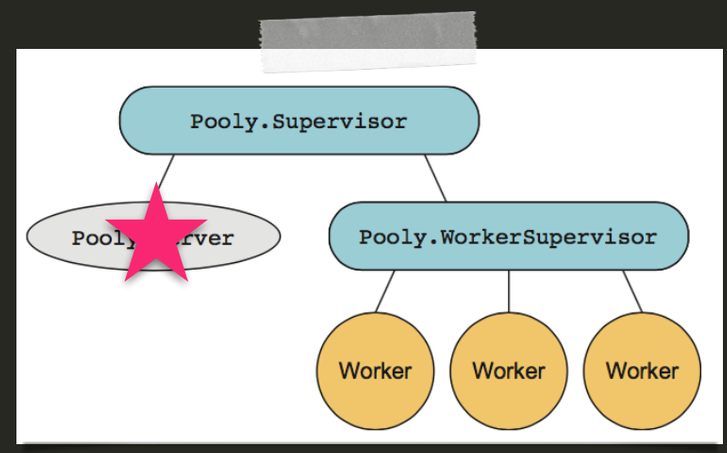
```elixir
defmodule Pooly.Server do
  defstruct State do
    sup: nil, worker_sup: nil, size: nil, workers: nil, mfa: nil
  end

  def handle_info(:start_worker_supervisor, state) do
    %{sup: sup, mfa: mfa, size: size} = state
    {:ok, worker_sup} = Supervisor.start_child(sup, supervisor_spec(mfa))
    workers = prepopulate(size, worker_sup)
    {:noreply, %{state | worker_sup: worker_sup, workers: workers}}
  end

  defp prepopulate(size, sup) when size > 1 do
    1..size |> Enum.map(fn _ -> new_worker(sup) end)
  end
  defp prepopulate(_size, _sup), do: []

  defp new_worker(sup) do
    {:ok, worker} = Supervisor.start_child(sup, [[]])
    worker
  end

  defp supervisor_spec(mfa) do
    supervisor(Pooly.WorkerSupervisor, [mfa],
               [restart: :temporary])
  end
end
```



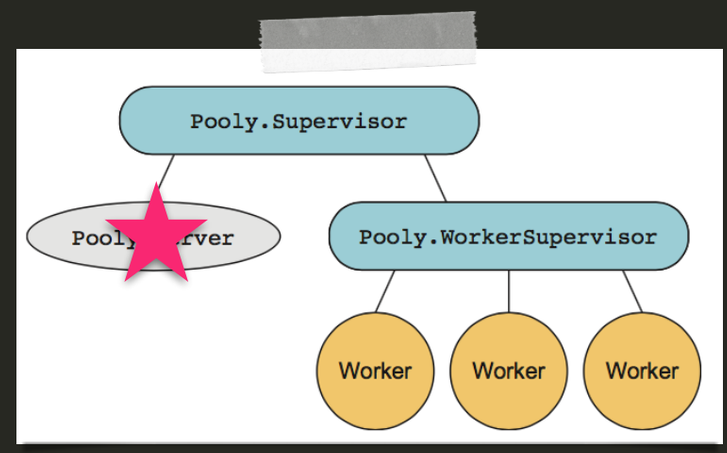
```elixir
defmodule Pooly.Server do
  defstruct State do
    sup: nil, worker_sup: nil, size: nil, workers: nil, mfa: nil
  end

  def handle_info(:start_worker_supervisor, state) do
    %{sup: sup, mfa: mfa, size: size} = state
    {:ok, worker_sup} = Supervisor.start_child(sup, supervisor_spec(mfa))
    workers = prepopulate(size, worker_sup)
    {:noreply, %{state | worker_sup: worker_sup, workers: workers}}
  end

  defp prepopulate(size, sup) when size > 1 do
    1..size |> Enum.map(fn _ -> new_worker(sup) end)
  end
  defp prepopulate(_size, _sup), do: []

  defp new_worker(sup) do
    {:ok, worker} = Supervisor.start_child(sup, [[]])
    worker
  end

  defp supervisor_spec(mfa) do
    supervisor(Pooly.WorkerSupervisor, [mfa],
               [restart: :temporary])
  end
end
```



```elixir
defmodule Pooly.Server do
  defstruct State do
    sup: nil, worker_sup: nil, size: nil, workers: nil, mfa: nil
  end

  def handle_info(:start_worker_supervisor, state) do
    %{sup: sup, mfa: mfa, size: size} = state
    {:ok, worker_sup} = Supervisor.start_child(sup, supervisor_spec(mfa))
    workers = prepopulate(size, worker_sup)
    {:noreply, %{state | worker_sup: worker_sup, workers: workers}}
  end

  defp prepopulate(size, sup) when size > 1 do
    1..size |> Enum.map(fn _ -> new_worker(sup) end)
  end
  defp prepopulate(_size, _sup), do: []

  defp new_worker(sup) do
    {:ok, worker} = Supervisor.start_child(sup, [[]])
    worker
  end

  defp supervisor_spec(mfa) do
    supervisor(Pooly.WorkerSupervisor, [mfa],
               [restart: :temporary])
  end
end
```

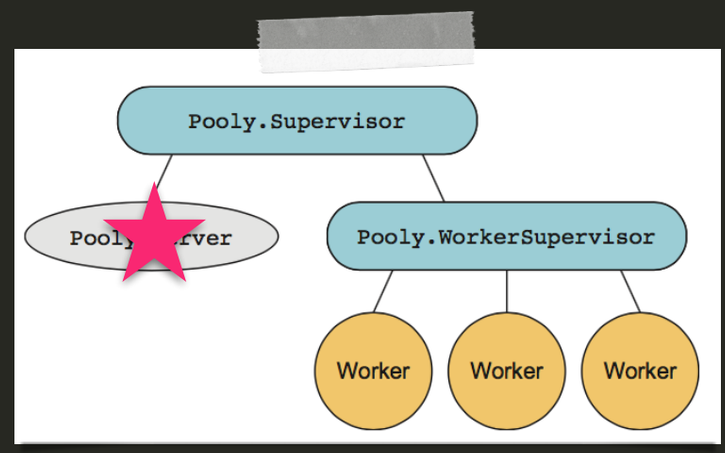


```elixir
defmodule Pooly.Server do
  defstruct State do
    sup: nil, worker_sup: nil, size: nil, workers: nil, mfa: nil
  end

  def handle_info(:start_worker_supervisor, state) do
    %{sup: sup, mfa: mfa, size: size} = state
    {:ok, worker_sup} = Supervisor.start_child(sup, supervisor_spec(mfa))
    workers = prepopulate(size, worker_sup)
    {:noreply, %{state | worker_sup: worker_sup, workers: workers}}
  end

  defp prepopulate(size, sup) when size > 1 do
    1..size |> Enum.map(fn _ -> new_worker(sup) end)
  end
  defp prepopulate(_size, _sup), do: []

  defp new_worker(sup) do
    {:ok, worker} = Supervisor.start_child(sup, [[]])
    worker
  end

  defp supervisor_spec(mfa) do
    supervisor(Pooly.WorkerSupervisor, [mfa],
               [restart: :temporary])
  end
end
```



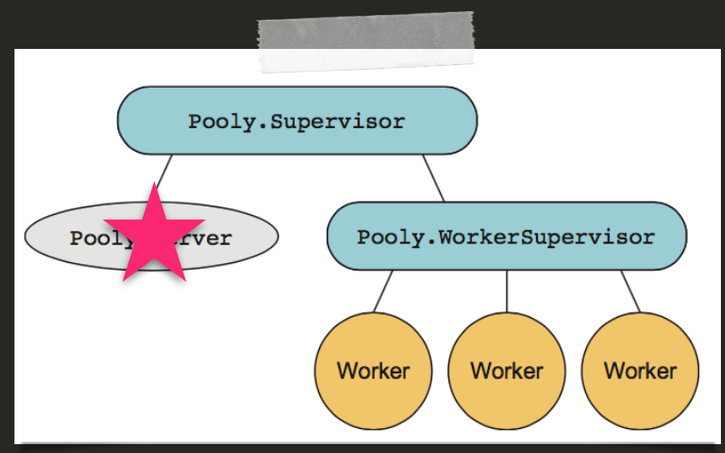
WORKER CHECK-OUT

```
defmodule Pooly.Server do

  def handle_call(:checkout, {from_pid, _ref}, state) do
    %{workers: workers, monitors: monitors} = state
    case workers do
      [worker|rest] ->
        ref = Process.monitor(from_pid)
        true = :ets.insert(monitors, {worker, ref})
        {:reply, worker, %{state | workers: rest}}

      [] ->
        {:reply, :noproc, state}
    end
  end

end
```



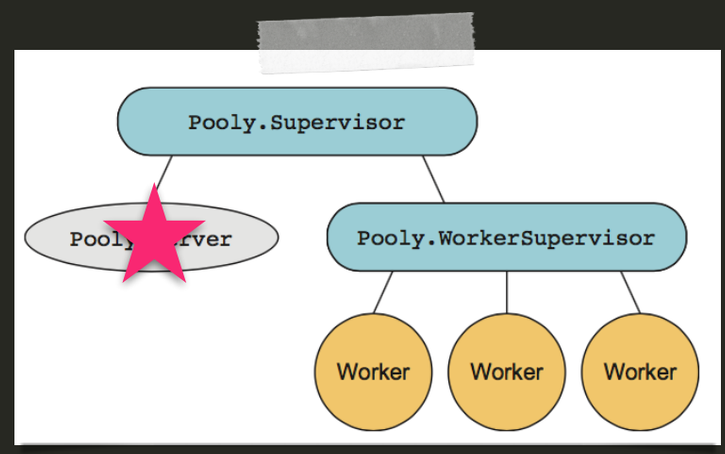
WORKER CHECK-OUT

```
defmodule Pooly.Server do

  def handle_call(:checkout, {from_pid, _ref}, state) do
    %{workers: workers, monitors: monitors} = state
    case workers do
      [worker|rest] ->
        ref = Process.monitor(from_pid)
        true = :ets.insert(monitors, {worker, ref})
        {:reply, worker, %{state | workers: rest}}

      [] ->
        {:reply, :noproc, state}
    end
  end

end
```



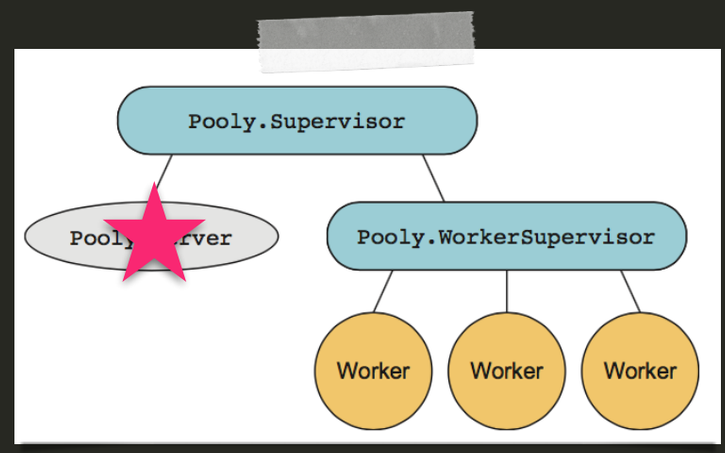
WORKER CHECK-OUT

```elixir
defmodule Pooly.Server do

  def handle_call(:checkout, {from_pid, _ref}, state) do
    %{workers: workers, monitors: monitors} = state
    case workers do
      [worker|rest] ->
        ref = Process.monitor(from_pid)
        true = :ets.insert(monitors, {worker, ref})
        {:reply, worker, %{state | workers: rest}}

      [] ->
        {:reply, :noproc, state}
    end
  end

end
```



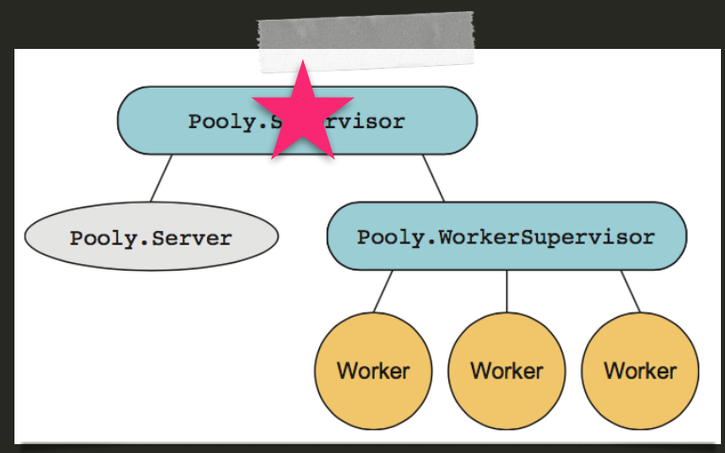
WORKER CHECK-IN

```
defmodule Pooly.Server do

  def handle_cast({:checkin, worker}, state) do
    %{workers: workers, monitors: monitors} = state
    case :ets.lookup(monitors, worker) do
      [{pid, ref}] ->
        true = Process.demonitor(ref)
        true = :ets.delete(monitors, pid)
        {:noreply, %{state | workers: [pid|workers]}}
      [] ->
        {:noreply, state}
    end
  end

end
```



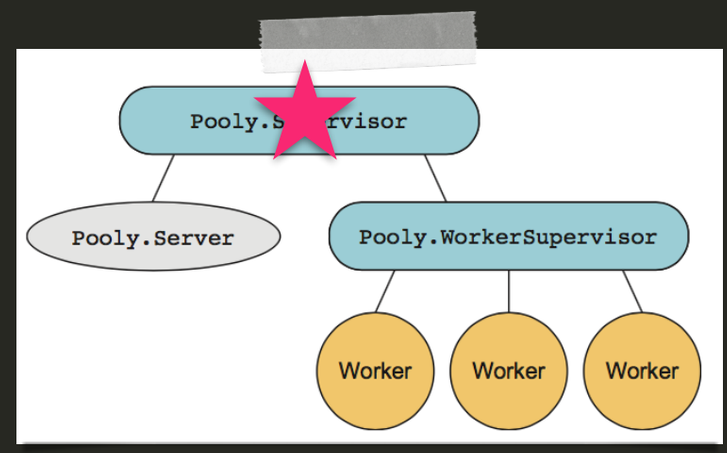
TOP-LEVEL SUPERVISOR

```
defmodule Pooly.Supervisor do
  use Supervisor

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

  def init(pool_config) do
    children = [
      worker(Pooly.Server, [self, pool_config])
    ]

    opts = [strategy: :one_for_all]

    supervise(children, opts)
  end
end
```



TOP-LEVEL SUPERVISOR

```elixir
defmodule Pooly.Supervisor do
  use Supervisor

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

  def init(pool_config) do
    children = [
      worker(Pooly.Server, [self, pool_config])
    ]

    opts = [strategy: :one_for_all]

    supervise(children, opts)
  end
end
```



VERSION 1

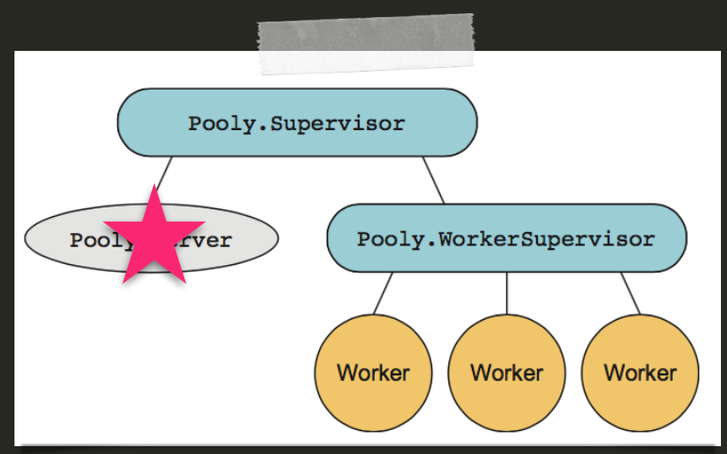
TYPE OF POOL	Single	Multiple
CREATION OF WORKERS	Fixed	Dynamic
CONSUMER RECOVERY	No	Yes
WORKER RECOVERY	No	Yes
QUEUEING FOR BUSY WORKERS	No	Yes

VERSION 2

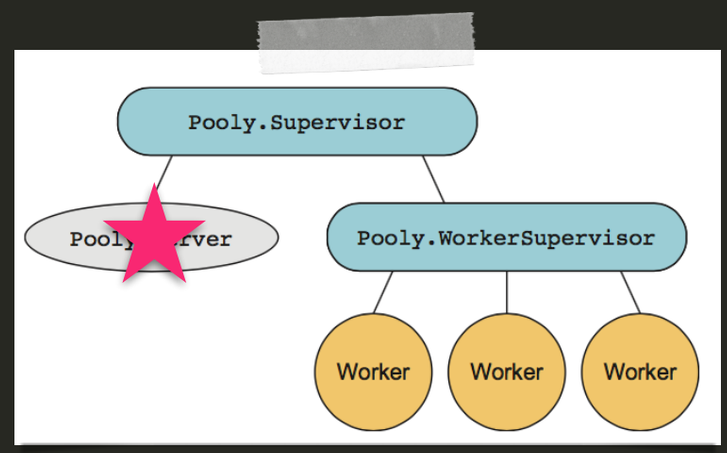
TYPE OF POOL	Single	Multiple
CREATION OF WORKERS	Fixed	Dynamic
CONSUMER RECOVERY	No	Yes
WORKER RECOVERY	No	Yes
QUEUEING FOR BUSY WORKERS	No	Yes

WHAT HAPPENS WHEN CONSUMER PROCESS DIES?

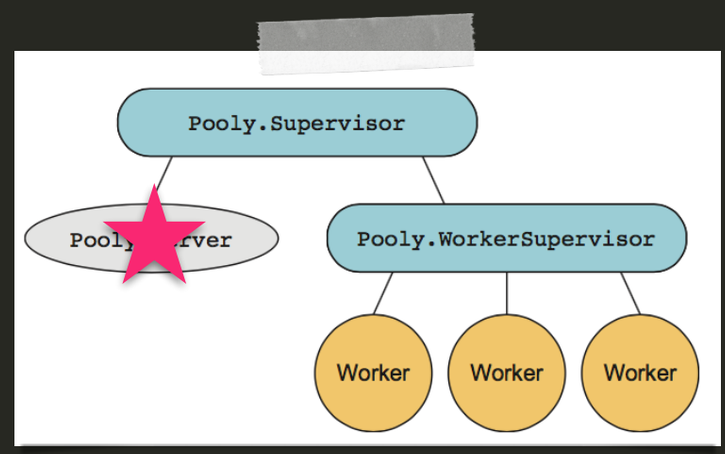
```elixir
defmodule Pooly.Server do
  def handle_info({:DOWN, ref, _, _, _}, state) do
    %{monitors: monitors, workers: workers} = state
    case :ets.match(monitors, {:"$1", ref}) do
      [[pid]] ->
        true = :ets.delete(monitors, pid)
        new_state = %{state | workers: [pid|workers]}
        {:noreply, new_state}
      [[]] ->
        {:noreply, state}
    end
  end
end
```



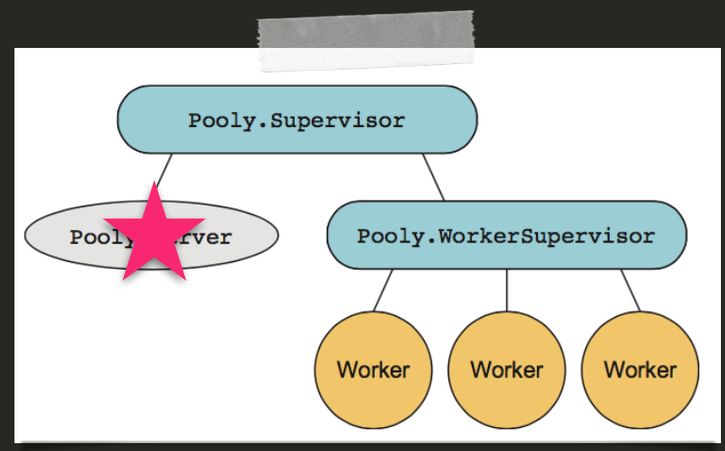
```elixir
defmodule Pooly.Server do
  def handle_info({:DOWN, ref, _, _, _}, state) do
    %{monitors: monitors, workers: workers} = state
    case :ets.match(monitors, {:"$1", ref}) do
      [[pid]] ->
        true = :ets.delete(monitors, pid)
        new_state = %{state | workers: [pid|workers]}
        {:noreply, new_state}
      [[]] ->
        {:noreply, state}
    end
  end
end
```



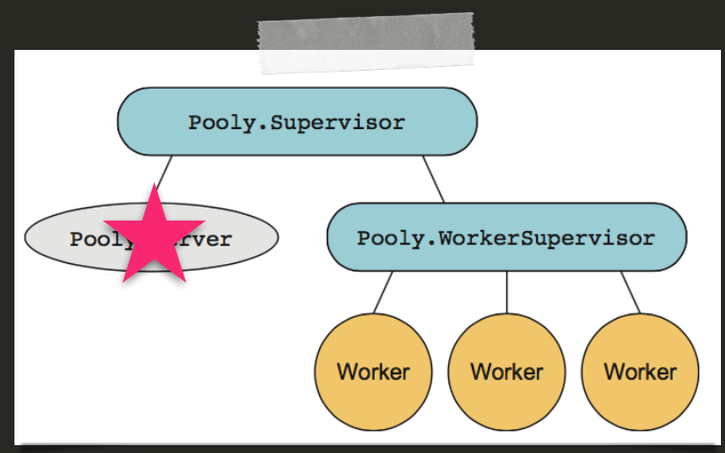
```elixir
defmodule Pooly.Server do
  def handle_info({:DOWN, ref, _, _, _}, state) do
    %{monitors: monitors, workers: workers} = state
    case :ets.match(monitors, {:"$1", ref}) do
      [[pid]] ->
        true = :ets.delete(monitors, pid)
        new_state = %{state | workers: [pid|workers]}
        {:noreply, new_state}
      [[]] ->
        {:noreply, state}
    end
  end
end
```



```elixir
defmodule Pooly.Server do
  def handle_info({:DOWN, ref, _, _, _}, state) do
    %{monitors: monitors, workers: workers} = state
    case :ets.match(monitors, {:"$1", ref}) do
      [[pid]] ->
        true = :ets.delete(monitors, pid)
        new_state = %{state | workers: [pid|workers]}
        {:noreply, new_state}

      [[]] ->
        {:noreply, state}
    end
  end
end
```

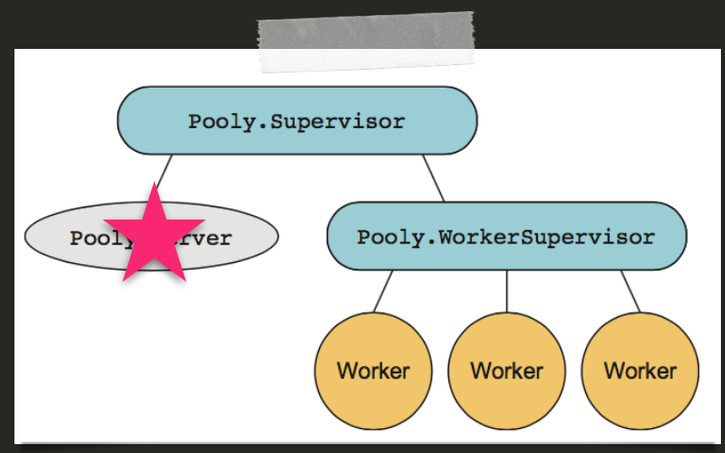


```elixir
defmodule Pooly.Server do
  def handle_info({:DOWN, ref, _, _, _}, state) do
    %{monitors: monitors, workers: workers} = state
    case :ets.match(monitors, {:"$1", ref}) do
      [[pid]] ->
        true = :ets.delete(monitors, pid)
        new_state = %{state | workers: [pid|workers]}
        {:noreply, new_state}
      [[]] ->
        {:noreply, state}
    end
  end
end
```

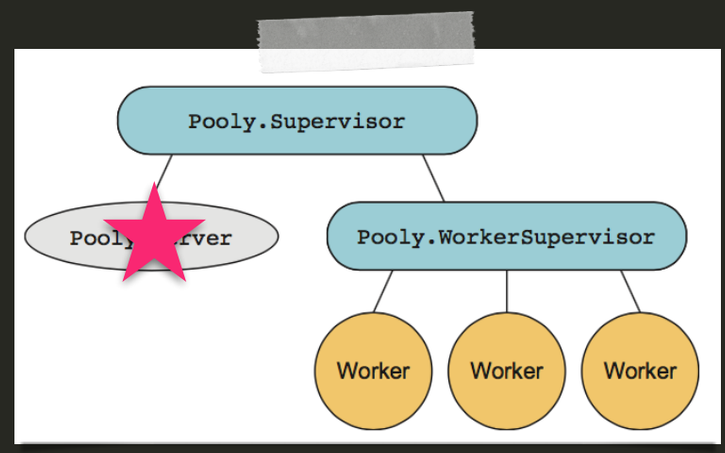


WHAT HAPPENS WHEN WORKER PROCESS DIES?

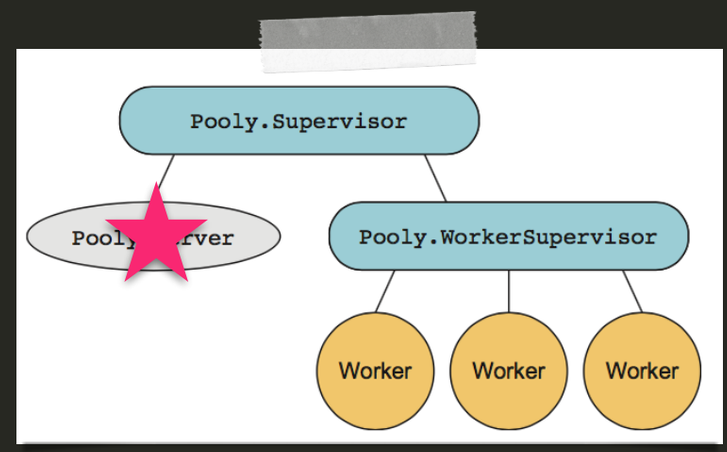
```elixir
defmodule Pooly.Server do
  def init([sup, pool_config]) when is_pid(sup) do
    Process.flag(:trap_exit, true)
    monitors = :ets.new(:monitors, [:private])
    init(pool_config, %State{sup: sup, monitors: monitors})
  end
end
```



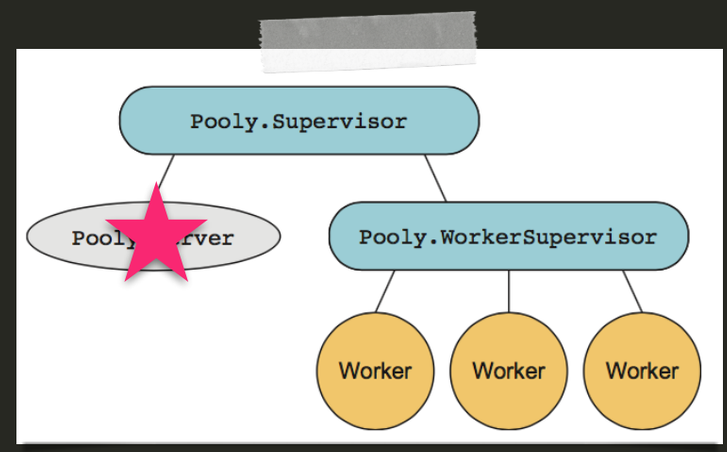
```elixir
defmodule Pooly.Server do
  def init([sup, pool_config]) when is_pid(sup) do
    Process.flag(:trap_exit, true)
    monitors = :ets.new(:monitors, [:private])
    init(pool_config, %State{sup: sup, monitors: monitors})
  end
end
```



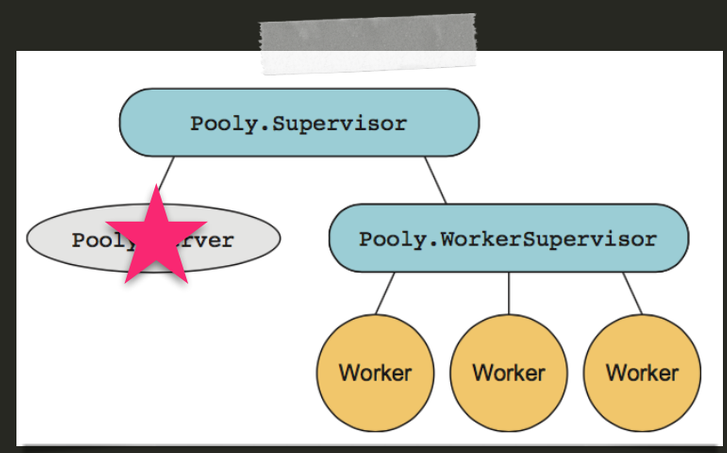
```elixir
def handle_info({:EXIT, pid, _reason}, state) do
  %{monitors: monitors,
    workers: workers,
    worker_sup: worker_sup} = state

  case :ets.lookup(monitors, pid) do
    [{pid, ref}] ->
      true = Process.demonitor(ref)
      true = :ets.delete(monitors, pid)
      new_state = %{state | workers: [new_worker(worker_sup)|workers]}
      {:noreply, new_state}

    [] ->
      {:noreply, state}
  end
end
```



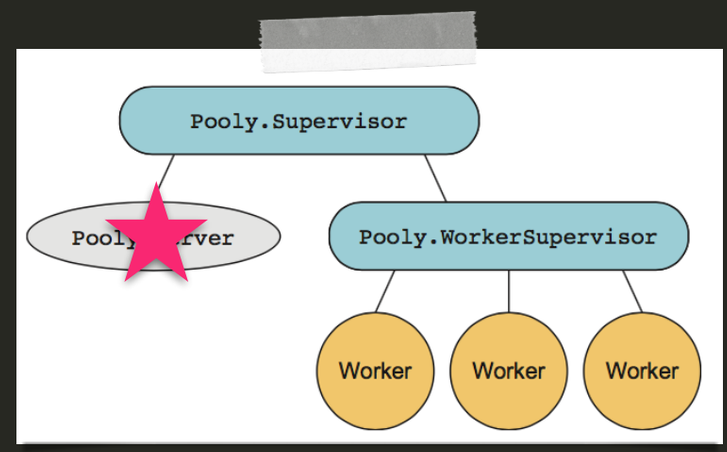
```elixir
def handle_info({:EXIT, pid, _reason}, state) do
  %{monitors: monitors,
    workers: workers,
    worker_sup: worker_sup} = state

  case :ets.lookup(monitors, pid) do
    [{pid, ref}] ->
      true = Process.demonitor(ref)
      true = :ets.delete(monitors, pid)
      new_state = %{state | workers: [new_worker(worker_sup)|workers]}
      {:noreply, new_state}

    [] ->
      {:noreply, state}
  end
end
```



```elixir
def handle_info({:EXIT, pid, _reason}, state) do
  %{monitors: monitors,
    workers: workers,
    worker_sup: worker_sup} = state

  case :ets.lookup(monitors, pid) do
    [{pid, ref}] ->
      true = Process.demonitor(ref)
      true = :ets.delete(monitors, pid)
      new_state = %{state | workers: [new_worker(worker_sup)|workers]}
      {:noreply, new_state}

    [] ->
      {:noreply, state}
  end
end
```



```elixir
def handle_info({:EXIT, pid, _reason}, state) do
  %{monitors: monitors,
    workers: workers,
    worker_sup: worker_sup} = state

  case :ets.lookup(monitors, pid) do
    [{pid, ref}] ->
      true = Process.demonitor(ref)
      true = :ets.delete(monitors, pid)
      new_state = %{state | workers: [new_worker(worker_sup)|workers]}
      {:noreply, new_state}

    [] ->
      {:noreply, state}
  end
end
```




ERROR
RECOVERY
FTW!!!
4GIFs.com

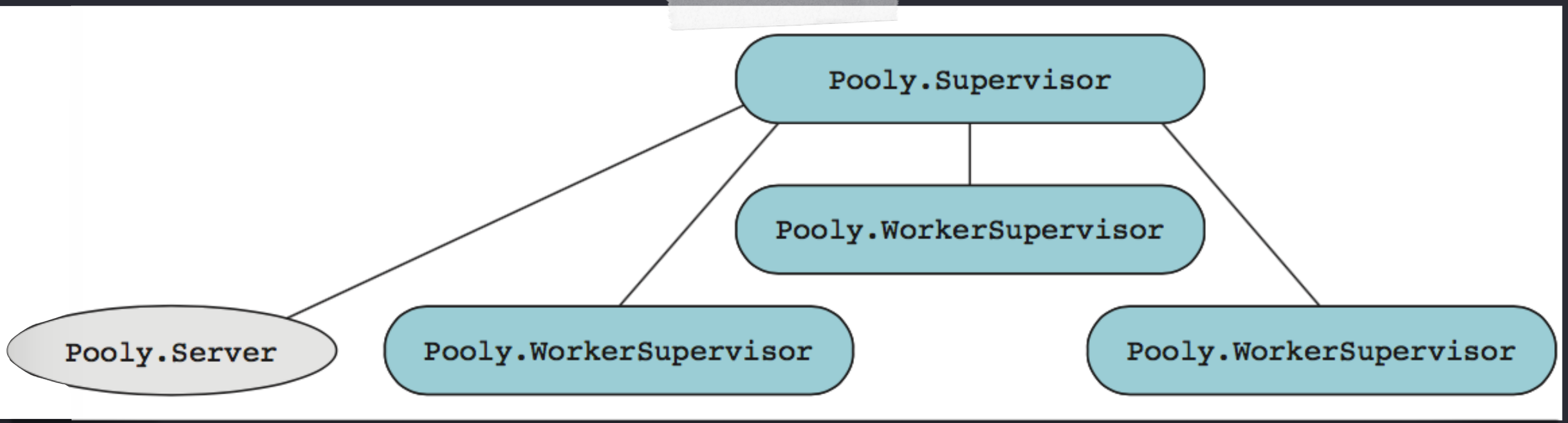
VERSION 2

TYPE OF POOL	Single	Multiple
CREATION OF WORKERS	Fixed	Dynamic
CONSUMER RECOVERY	No	Yes
WORKER RECOVERY	No	Yes
QUEUEING FOR BUSY WORKERS	No	Yes

VERSION 3

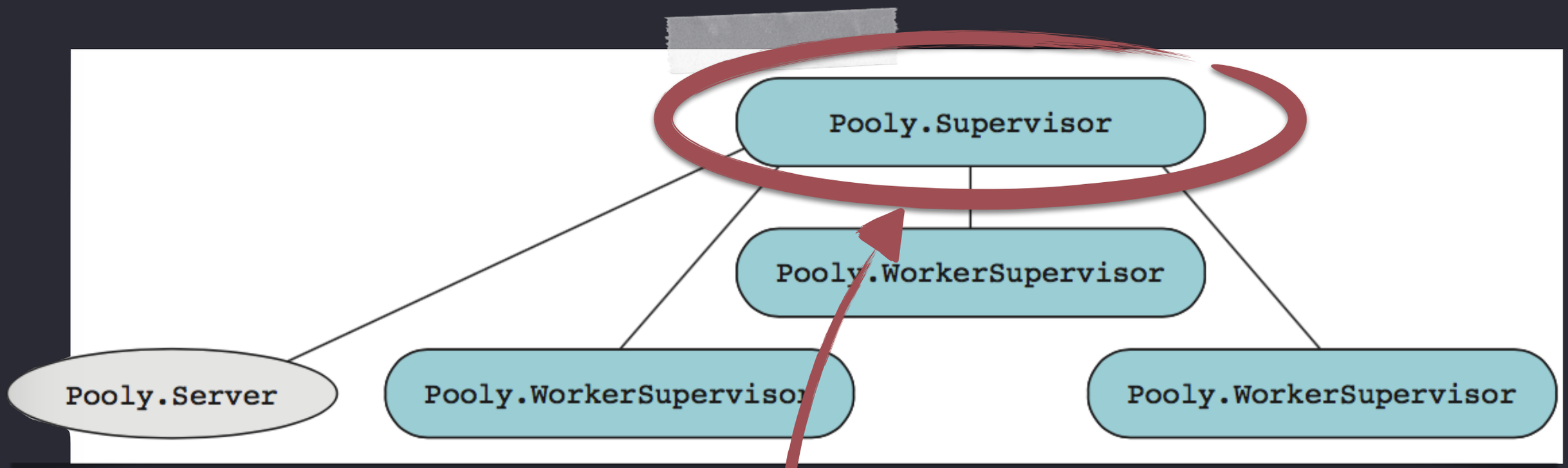
TYPE OF POOL	Single	Multiple
CREATION OF WORKERS	Fixed	Dynamic
CONSUMER RECOVERY	No	Yes
WORKER RECOVERY	No	Yes
QUEUEING FOR BUSY WORKERS	No	Yes

ATTEMPT #1



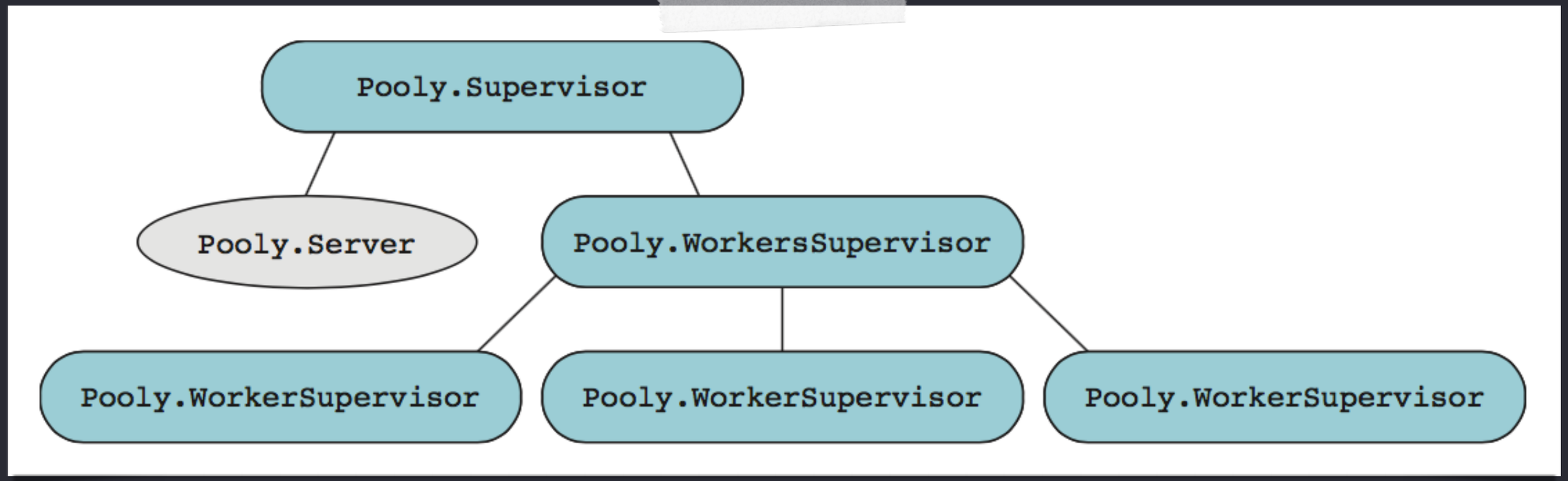
WHAT'S WRONG WITH THIS?

ATTEMPT #1



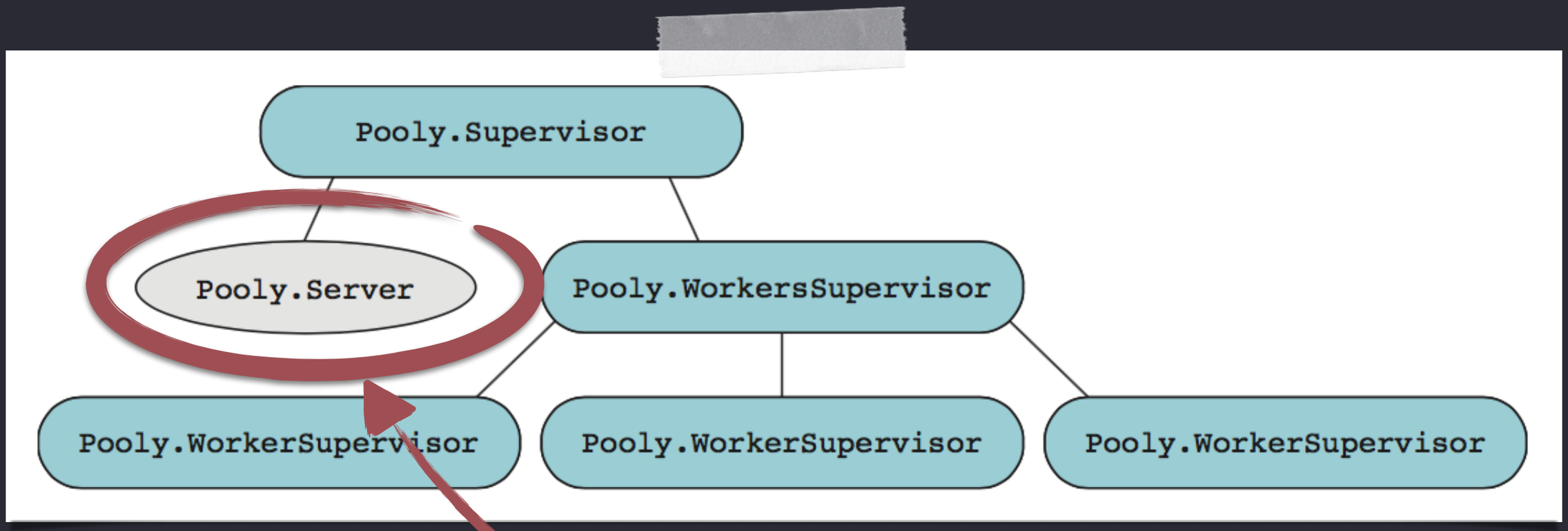
THIS.

ATTEMPT #2



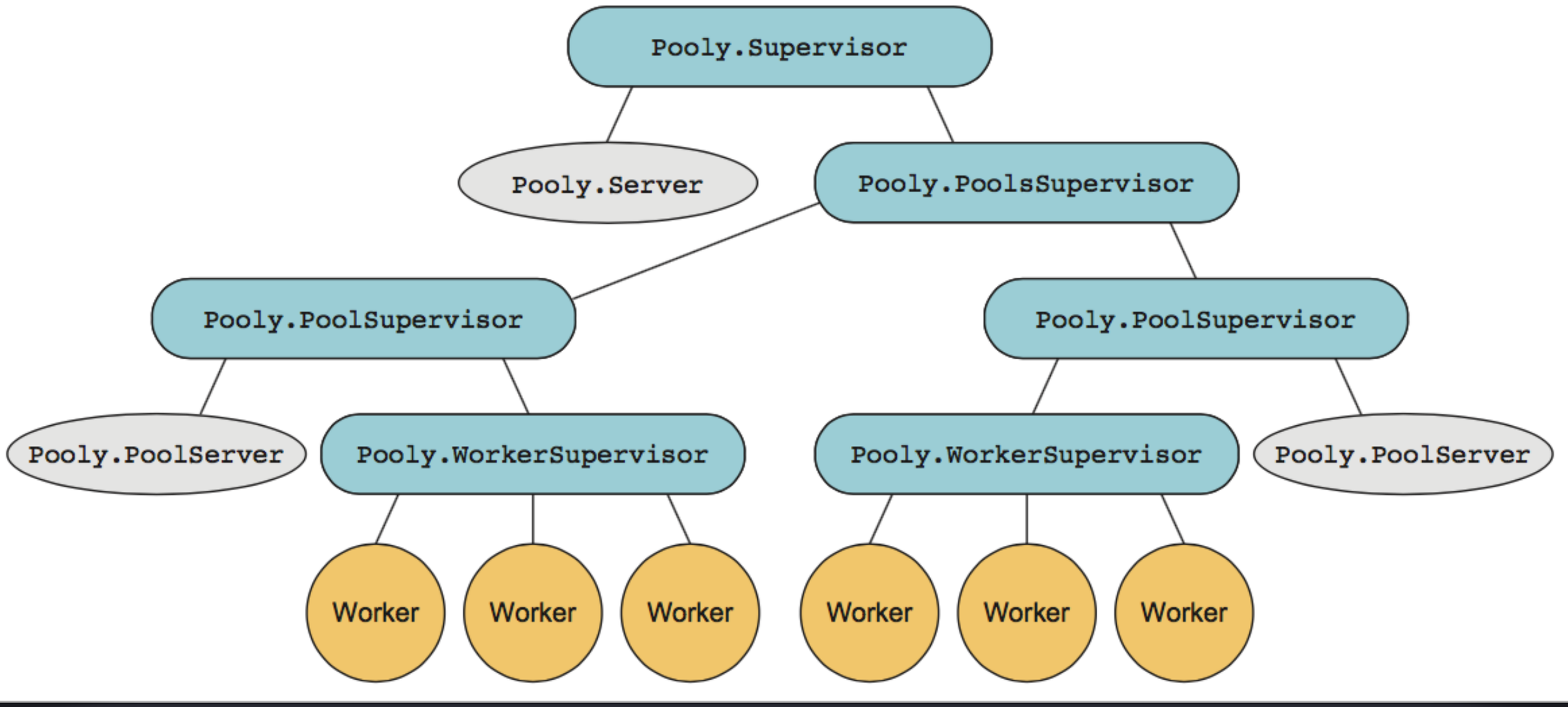
WHAT'S WRONG WITH THIS?

ATTEMPT #2



THIS.

ATTEMPT #3

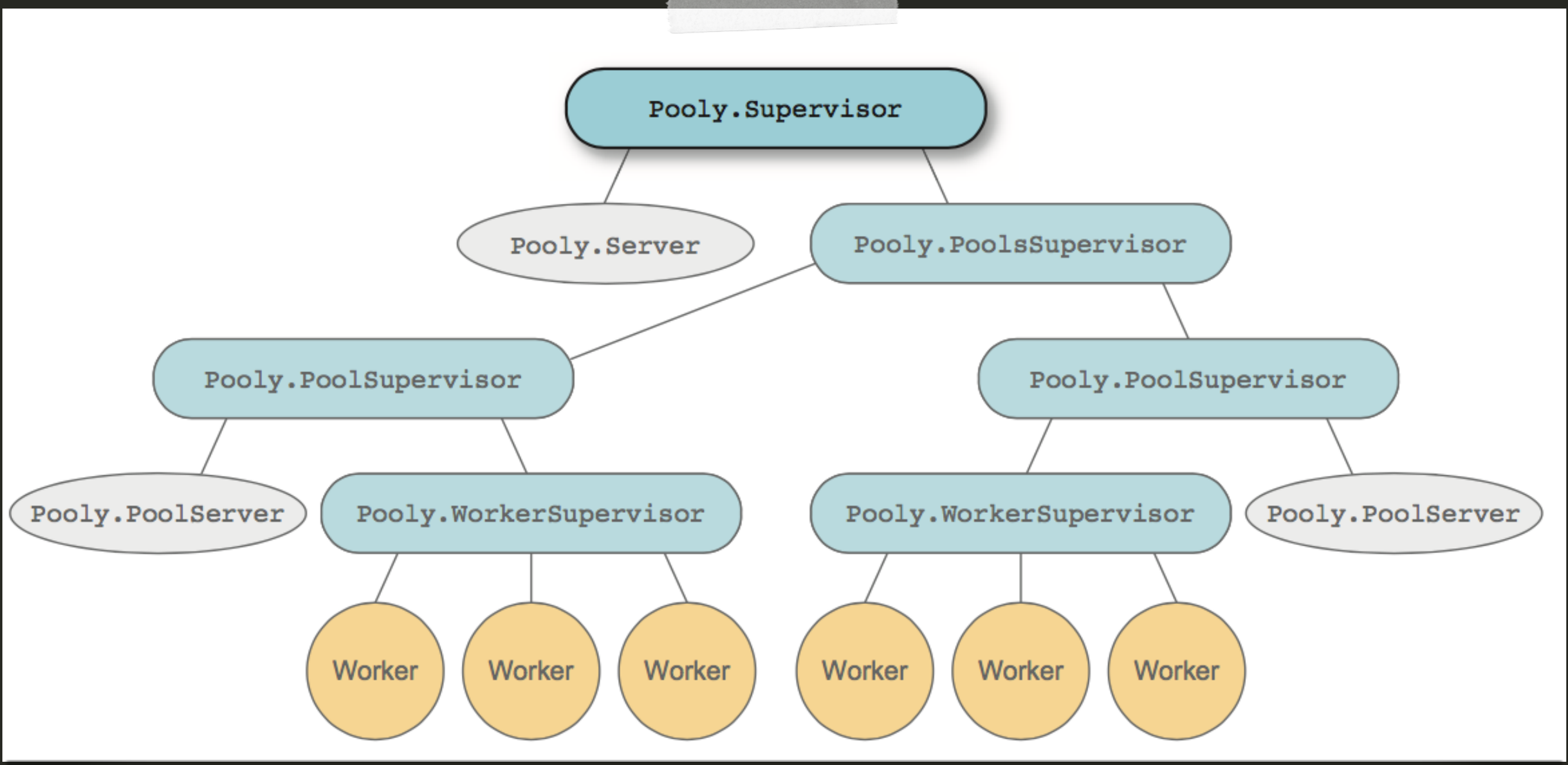

Pooly.Supervisor
Pooly.Server
Pooly.PoolsSupervisor
Pooly.PoolSupervisor
Pooly.PoolSupervisor
Pooly.PoolServer
Pooly.WorkerSupervisor
Pooly.WorkerSupervisor
Pooly.PoolServer
Worker
Worker
Worker
Worker
Worker
Worker

EACH POOL

GETS A NAME

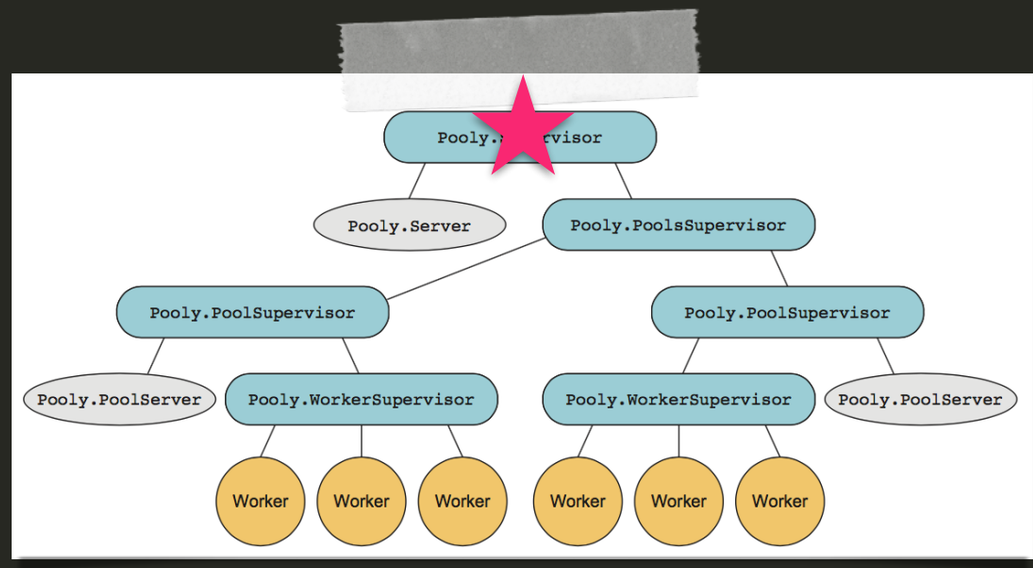
```elixir
defmodule Pooly do
  use Application

  def start(_type, _args) do
    pools_config =
      [
        [name: "Pool1",
          mfa: {SampleWorker, :start_link, []}, size: 2],
        [name: "Pool2",
          mfa: {SampleWorker, :start_link, []}, size: 3],
        [name: "Pool3",
          mfa: {SampleWorker, :start_link, []}, size: 4],
      ]

    start_pools(pools_config)
  end
end
```

ADDING THE TOP MOST TOP LEVEL SUPERVISOR



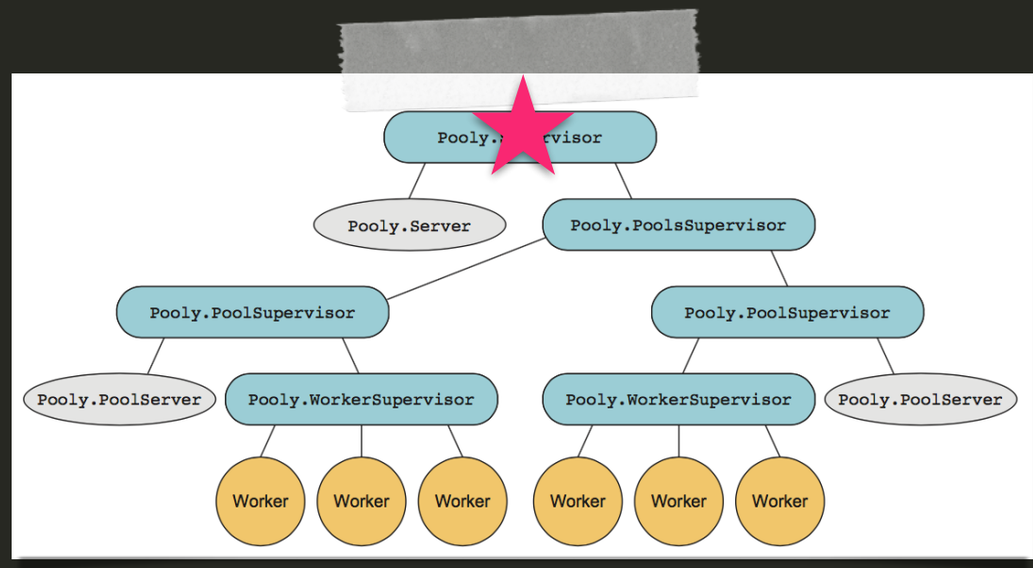
ADDING THE TOP LEVEL SUPERVISOR

```elixir
defmodule Pooly.Supervisor do
  use Supervisor

  def start_link(pools_config) do
    Supervisor.start_link(__MODULE__, pools_config, name: __MODULE__)
  end

  def init(pools_config) do
    children = [
      supervisor(Pooly.PoolsSupervisor, []),
      worker(Pooly.Server, [pools_config])
    ]
    opts = [strategy: :one_for_all]
    supervise(children, opts)
  end
end
```



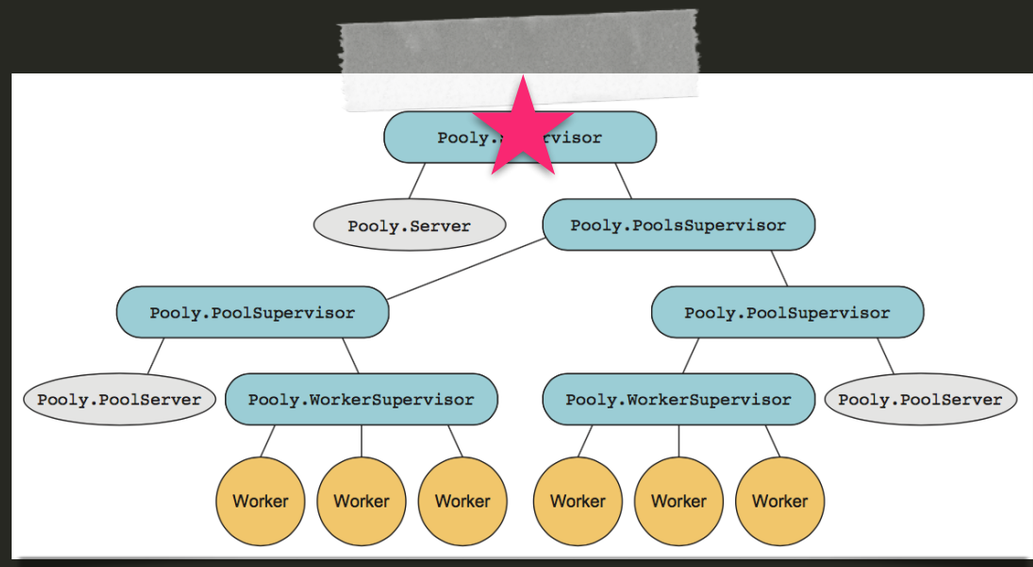
ADDING THE TOP LEVEL SUPERVISOR

```elixir
defmodule Pooly.Supervisor do
  use Supervisor

  def start_link(pools_config) do
    Supervisor.start_link(__MODULE__, pools_config, name: __MODULE__)
  end

  def init(pools_config) do
    children = [
      supervisor(Pooly.PoolsSupervisor, []),
      worker(Pooly.Server, [pools_config])
    ]
    opts = [strategy: :one_for_all]
    supervise(children, opts)
  end
end
```



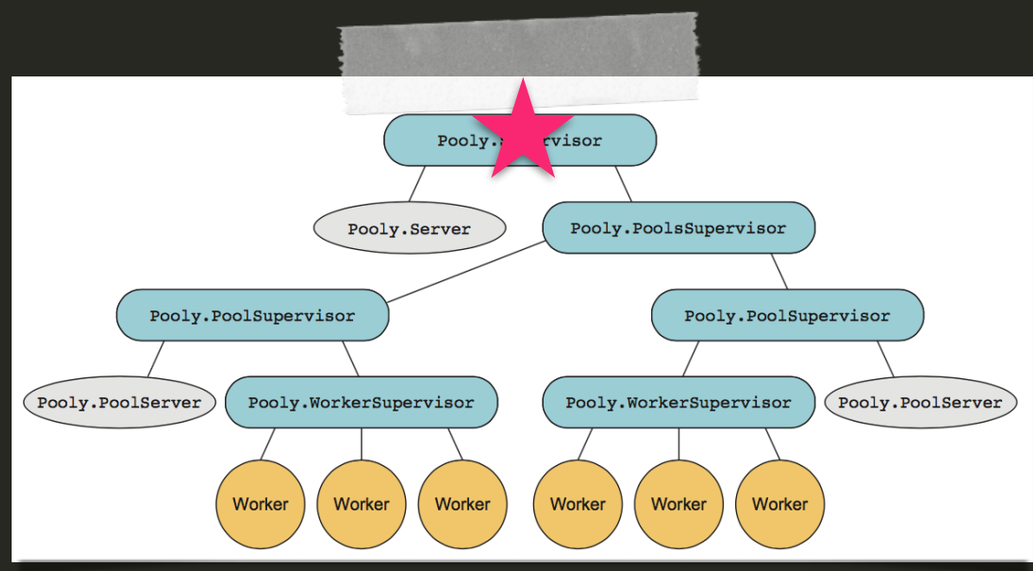
ADDING THE TOP LEVEL SUPERVISOR

```
defmodule Pooly.Supervisor do
  use Supervisor

  def start_link(pools_config) do
    Supervisor.start_link(__MODULE__, pools_config, name: __MODULE__)
  end

  def init(pools_config) do
    children = [
      supervisor(Pooly.PoolsSupervisor, []),
      worker(Pooly.Server, [pools_config])
    ]
    opts = [strategy: :one_for_all]
    supervise(children, opts)
  end
end
```

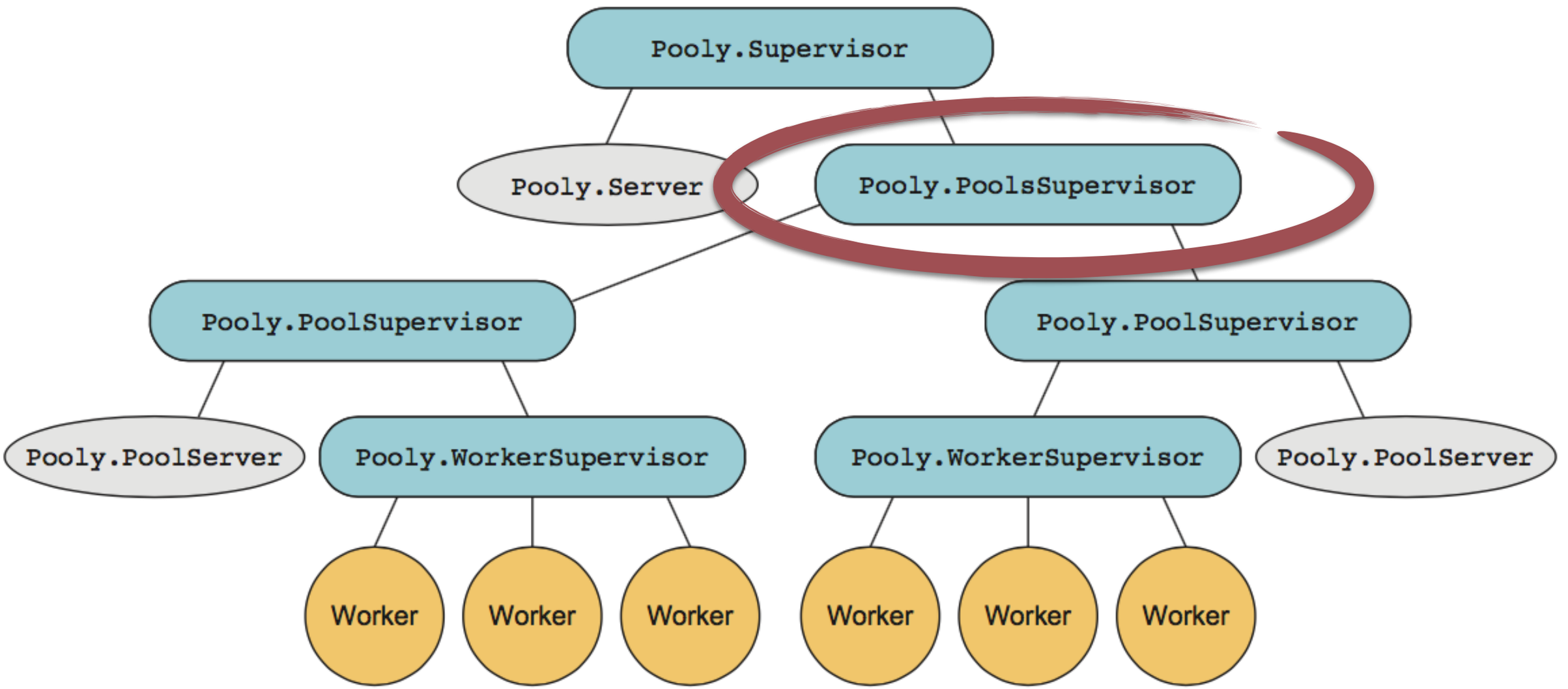


ADDING THE TOP LEVEL SUPERVISOR

```
defmodule Pooly.Supervisor do
  use Supervisor

  def start_link(pools_config) do
    Supervisor.start_link(__MODULE__, pools_config, name: __MODULE__)
  end

  def init(pools_config) do
    children = [
      supervisor(Pooly.PoolsSupervisor, []),
      worker(Pooly.Server, [pools_config])
    ]
    opts = [strategy: :one_for_all]
    supervise(children, opts)
  end
end
```

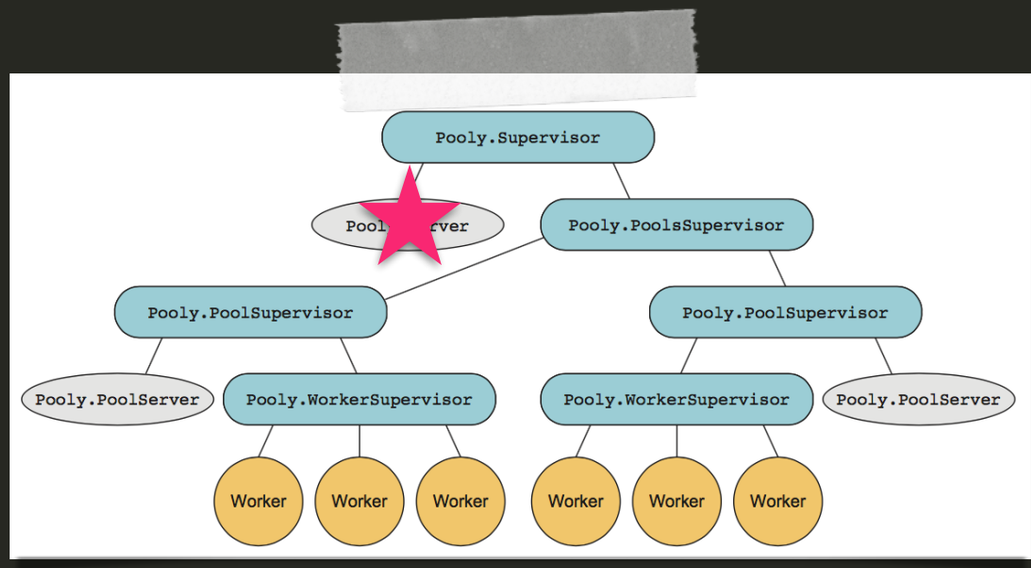


ADDING THE POOLS SUPERVISOR



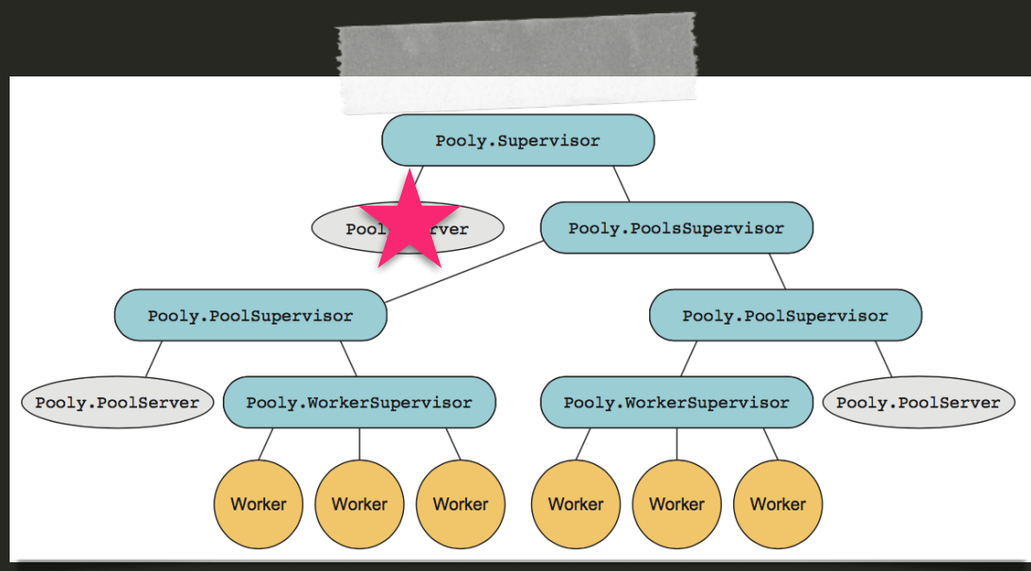
ADDING THE POOLS SUPERVISOR

```elixir
defmodule Pooly.PoolsSupervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, [], name: __MODULE__)
  end

  def init(_) do
    opts = [strategy: :one_for_one]

    supervise([], opts)
  end
end
```



ADDING THE POOLS SUPERVISOR

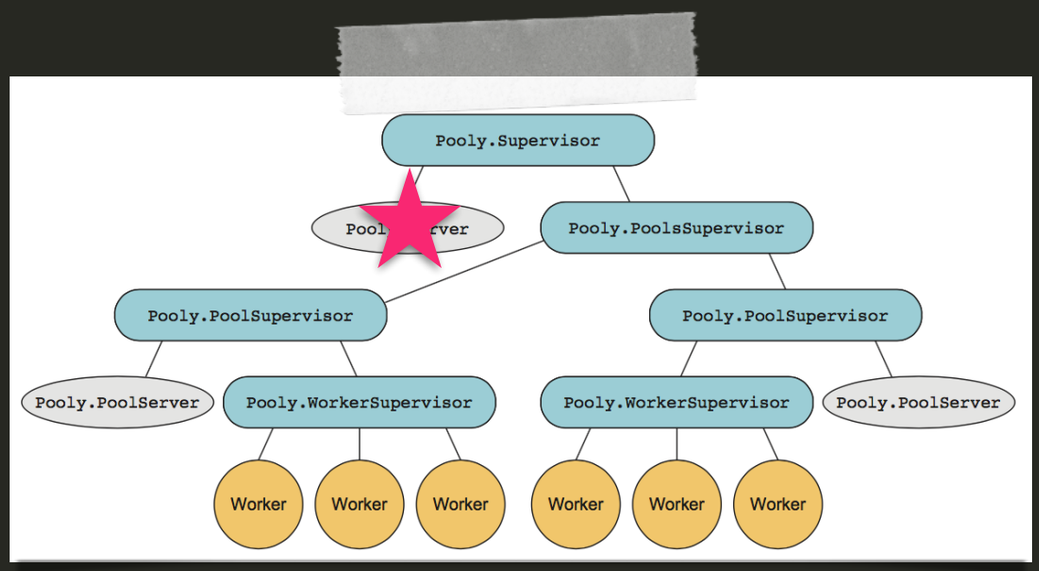
```elixir
defmodule Pooly.PoolsSupervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, [], name: __MODULE__)
  end

  def init(_) do
    opts = [strategy: :one_for_one]

    supervise([], opts)
  end
end
```



ADDING THE POOLS SUPERVISOR

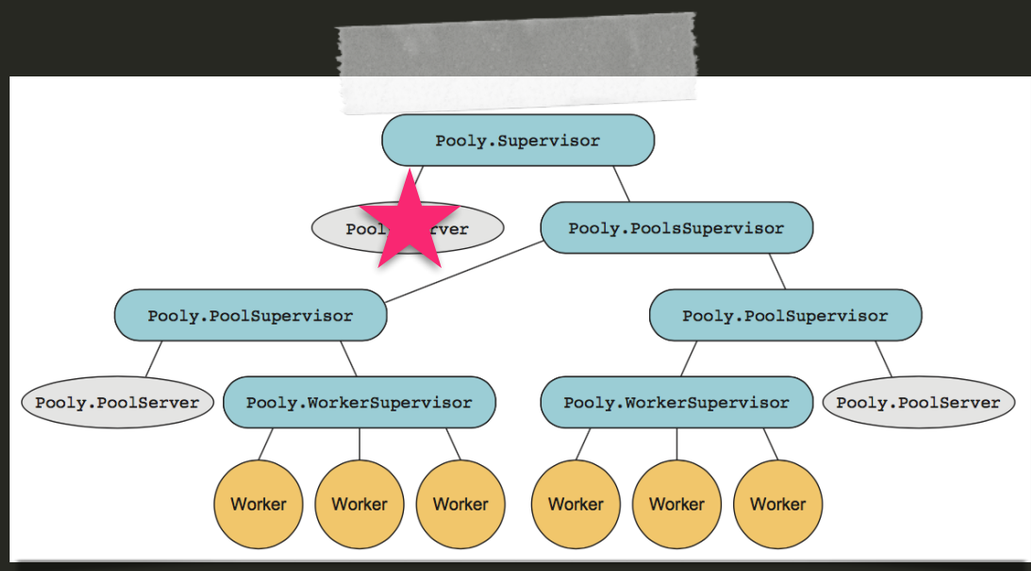
```elixir
defmodule Pooly.PoolsSupervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, [], name: __MODULE__)
  end

  def init(_) do
    opts = [strategy: :one_for_one]

    supervise([], opts)
  end
end
```

Empty child spec!

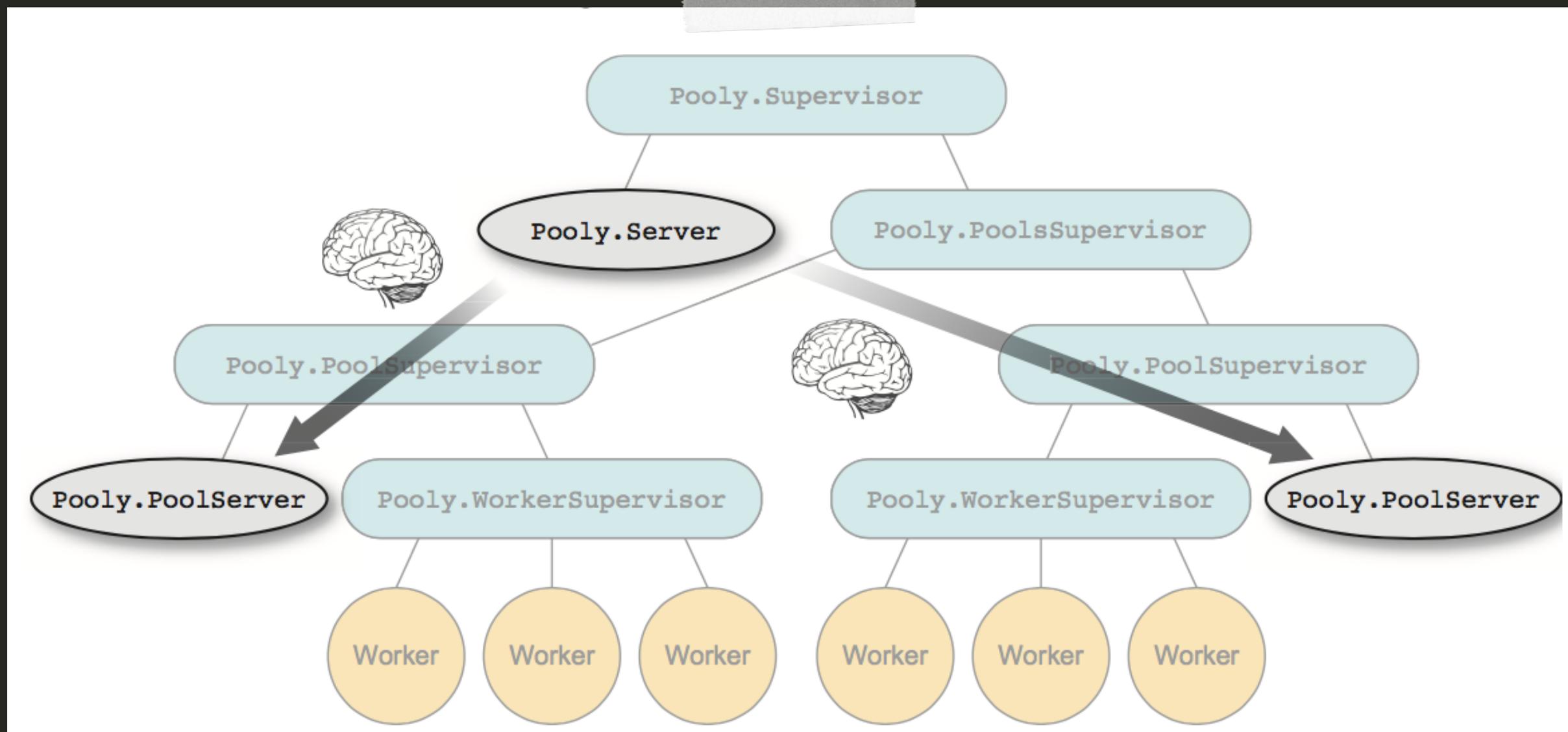


ADDING THE POOLS SUPERVISOR

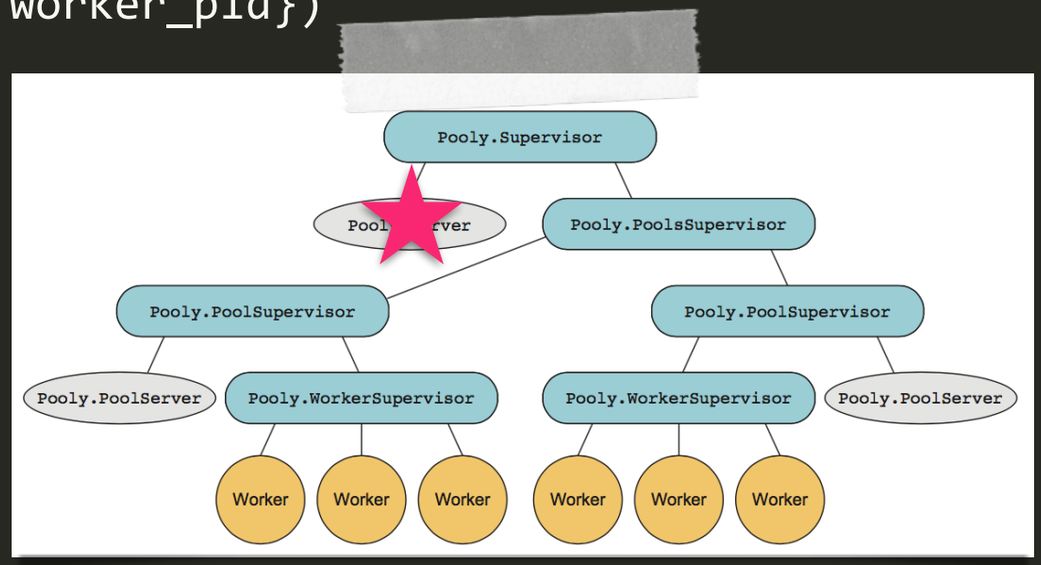
```elixir
defmodule Pooly.PoolsSupervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, [], name: __MODULE__)
  end

  def init(_) do
    opts = [strategy: :one_for_one]

    supervise([], opts)
  end
end
```



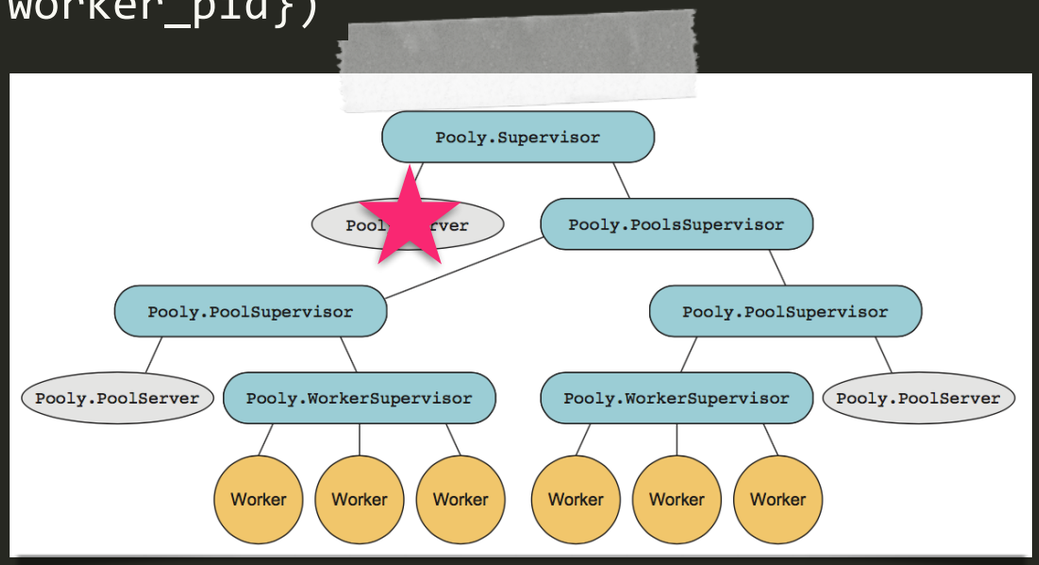
MAKING Pooly.Server DUMBER



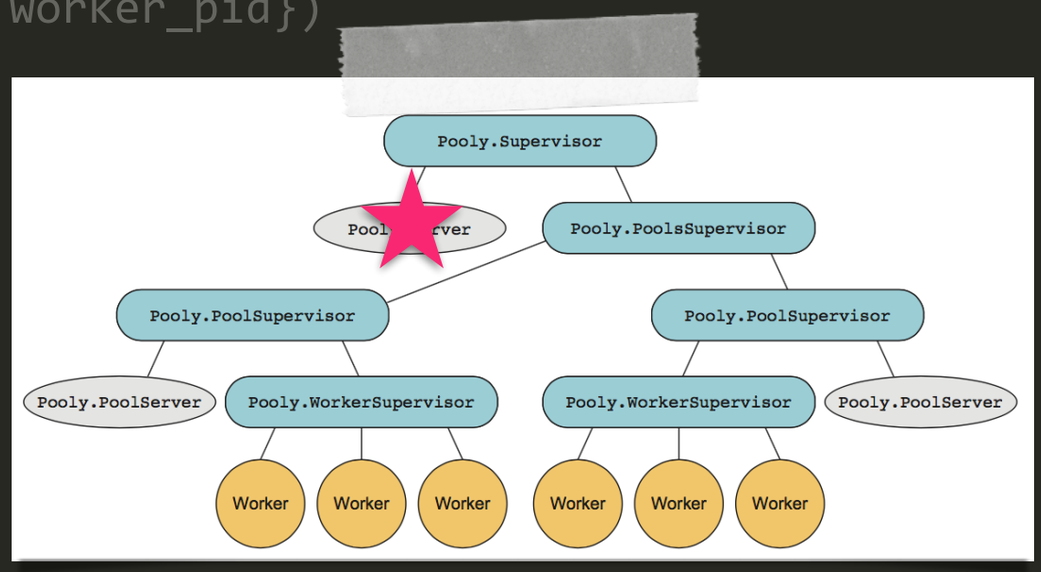
```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  def start_link(pools_config) do
    GenServer.start_link(__MODULE__, pools_config, name: __MODULE__)
  end

  def checkout(pool_name) do
    GenServer.call(:"#{pool_name}Server", :checkout)
  end

  def checkin(pool_name, worker_pid) do
    GenServer.cast(:"#{pool_name}Server", {:checkin, worker_pid})
  end

  def init(pools_config) do
    pools_config |> Enum.each(fn(pool_config) ->
      send(self, {:start_pool, pool_config})
    end)

    {:ok, pools_config}
  end

  def handle_info({:start_pool, pool_config}, state) do
    {:ok, _pool_sup} = Supervisor.start_child(Pooly.PoolsSupervisor,
                                              supervisor_spec(pool_config))
    {:noreply, state}
  end

  defp supervisor_spec(pool_config) do
    opts = [id: :"#{pool_config[:name]}Supervisor"]
    supervisor(Pooly.PoolSupervisor, [pool_config], opts)
  end
end
```



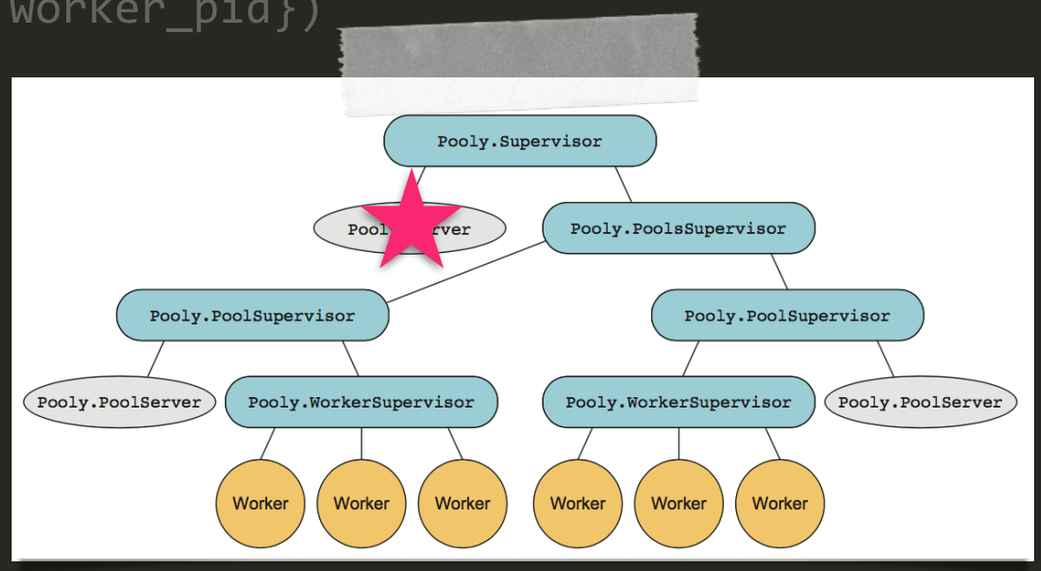
```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  def start_link(pools_config) do
    GenServer.start_link(__MODULE__, pools_config, name: __MODULE__)
  end

  def checkout(pool_name) do
    GenServer.call(:"#{pool_name}Server", :checkout)
  end

  def checkin(pool_name, worker_pid) do
    GenServer.cast(:"#{pool_name}Server", {:checkin, worker_pid})
  end

  def init(pools_config) do
    pools_config |> Enum.each(fn(pool_config) ->
      send(self, {:start_pool, pool_config})
    end)

    {:ok, pools_config}
  end

  def handle_info({:start_pool, pool_config}, state) do
    {:ok, _pool_sup} = Supervisor.start_child(Pooly.PoolsSupervisor,
                                              supervisor_spec(pool_config))
    {:noreply, state}
  end

  defp supervisor_spec(pool_config) do
    opts = [id: :"#{pool_config[:name]}Supervisor"]
    supervisor(Pooly.PoolSupervisor, [pool_config], opts)
  end
end
```



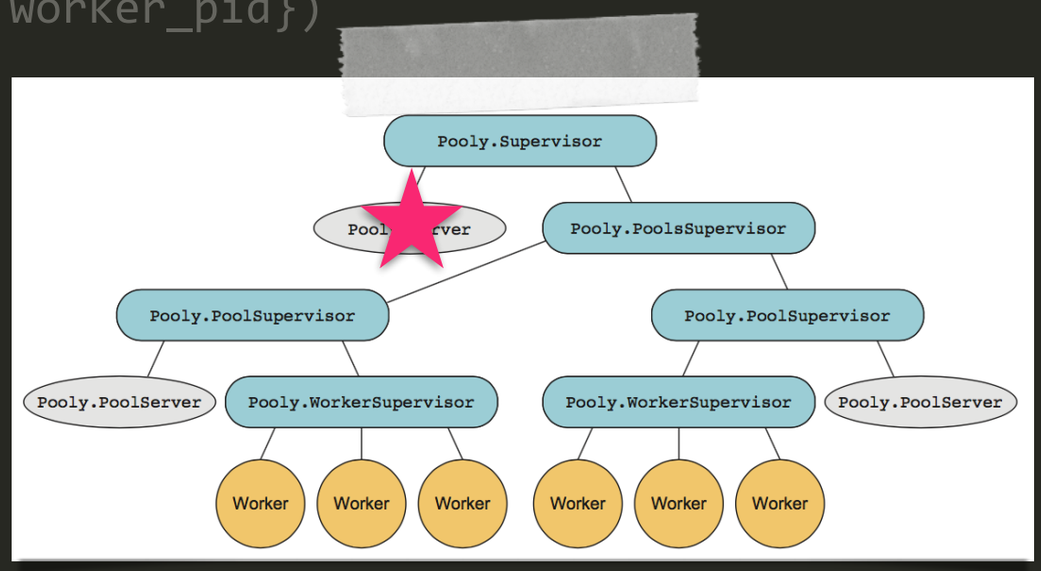
```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  def start_link(pools_config) do
    GenServer.start_link(__MODULE__, pools_config, name: __MODULE__)
  end

  def checkout(pool_name) do
    GenServer.call(:"#{pool_name}Server", :checkout)
  end

  def checkin(pool_name, worker_pid) do
    GenServer.cast(:"#{pool_name}Server", {:checkin, worker_pid})
  end

  def init(pools_config) do
    pools_config |> Enum.each(fn(pool_config) ->
      send(self, {:start_pool, pool_config})
    end)

    {:ok, pools_config}
  end

  def handle_info({:start_pool, pool_config}, state) do
    {:ok, _pool_sup} = Supervisor.start_child(Pooly.PoolsSupervisor,
                                              supervisor_spec(pool_config))
    {:noreply, state}
  end

  defp supervisor_spec(pool_config) do
    opts = [id: :"#{pool_config[:name]}Supervisor"]
    supervisor(Pooly.PoolSupervisor, [pool_config], opts)
  end
end
```



```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  def start_link(pools_config) do
    GenServer.start_link(__MODULE__, pools_config, name: __MODULE__)
  end

  def checkout(pool_name) do
    GenServer.call(:"#{pool_name}Server", :checkout)
  end

  def checkin(pool_name, worker_pid) do
    GenServer.cast(:"#{pool_name}Server", {:checkin, worker_pid})
  end

  def init(pools_config) do
    pools_config |> Enum.each(fn(pool_config) ->
      send(self, {:start_pool, pool_config})
    end)

    {:ok, pools_config}
  end

  def handle_info({:start_pool, pool_config}, state) do
    {:ok, _pool_sup} = Supervisor.start_child(Pooly.PoolsSupervisor,
                                              supervisor_spec(pool_config))
    {:noreply, state}
  end

  defp supervisor_spec(pool_config) do
    opts = [id: :"#{pool_config[:name]}Supervisor"]
    supervisor(Pooly.PoolSupervisor, [pool_config], opts)
  end
end
```

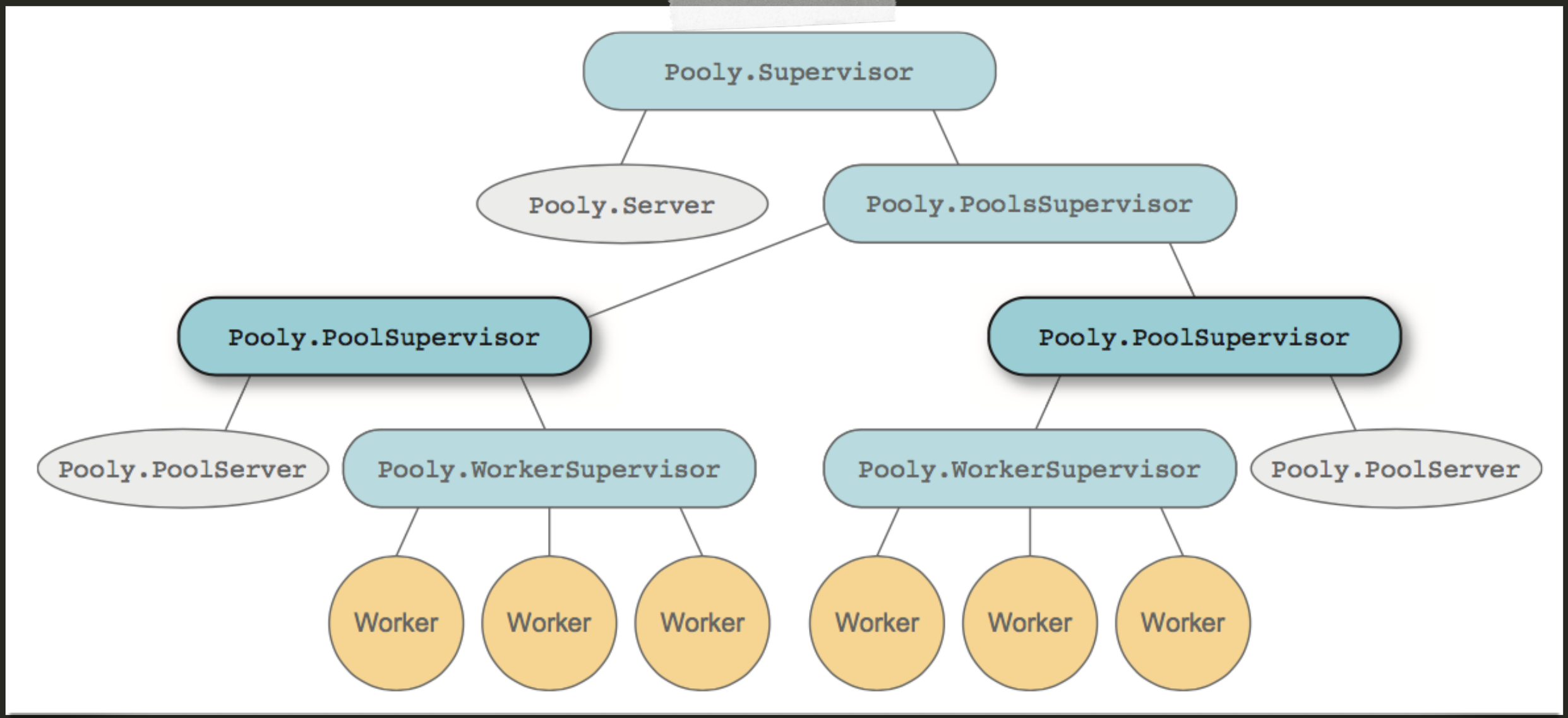


```elixir
defmodule Pooly.Server do
  use GenServer
  import Supervisor.Spec

  def start_link(pools_config) do
    GenServer.start_link(__MODULE__, pools_config, name: __MODULE__)
  end

  def checkout(pool_name) do
    GenServer.call(:"#{pool_name}Server", :checkout)
  end

  def checkin(pool_name, worker_pid) do
    GenServer.cast(:"#{pool_name}Server", {:checkin, worker_pid})
  end

  def init(pools_config) do
    pools_config |> Enum.each(fn(pool_config) ->
      send(self, {:start_pool, pool_config})
    end)

    {:ok, pools_config}
  end

  def handle_info({:start_pool, pool_config}, state) do
    {:ok, _pool_sup} = Supervisor.start_child(Pooly.PoolsSupervisor,
                                              supervisor_spec(pool_config))
    {:noreply, state}
  end

  defp supervisor_spec(pool_config) do
    opts = [id: :"#{pool_config[:name]}Supervisor"]
    supervisor(Pooly.PoolSupervisor, [pool_config], opts)
  end
end
```

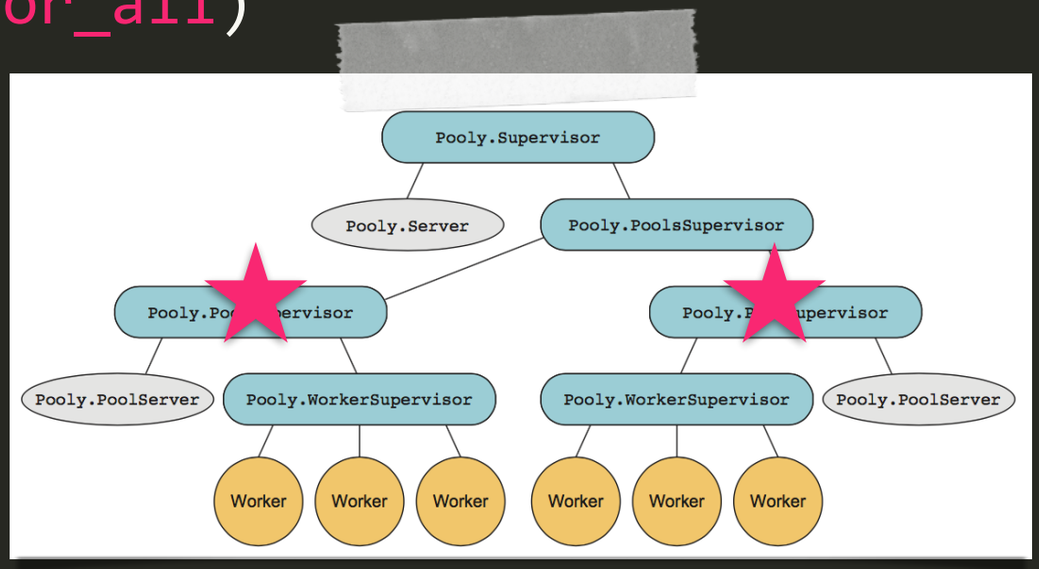


Unique spec ID!

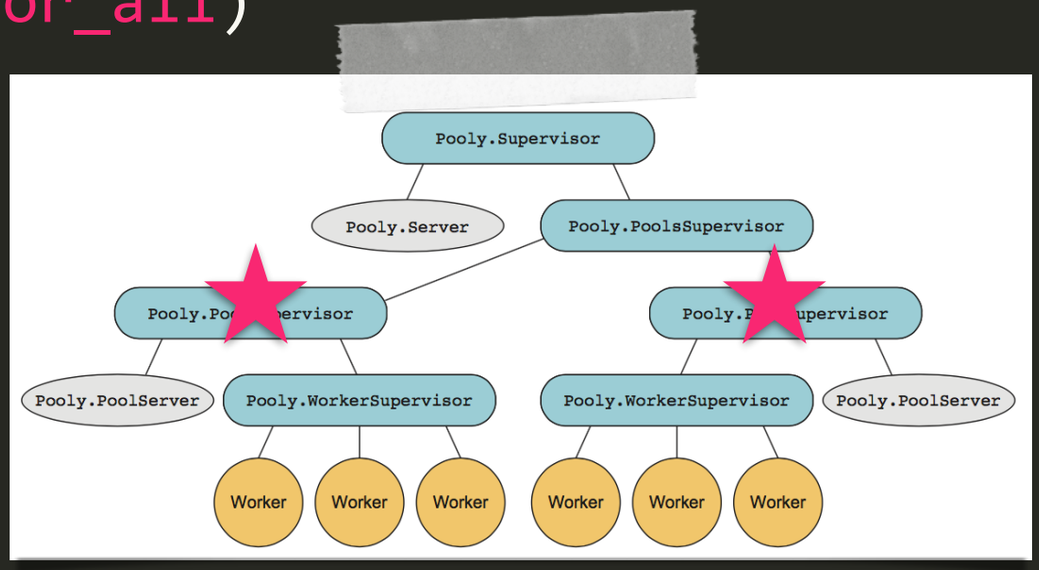
ADDING THE POOL SUPERVISOR



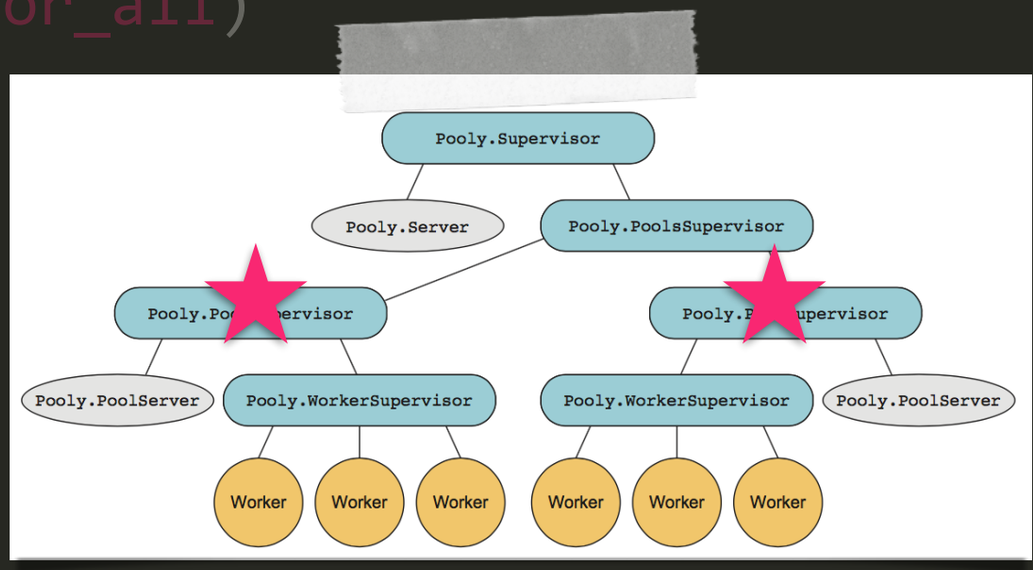
```elixir
defmodule Pooly.PoolSupervisor do
  use Supervisor

  def start_link(pool_config) do
    Supervisor.start_link(__MODULE__, pool_config,
                          name: :"#{pool_config[:name]}Supervisor")
  end

  def init(pool_config) do
    children = [
      worker(Pooly.PoolServer, [self, pool_config])
    ]

    supervise(children, strategy: :one_for_all)
  end
end
```



```elixir
defmodule Pooly.PoolSupervisor do
  use Supervisor

  def start_link(pool_config) do
    Supervisor.start_link(__MODULE__, pool_config,
                          name: :"#{pool_config[:name]}Supervisor")
  end

  def init(pool_config) do
    children = [
      worker(Pooly.PoolServer, [self, pool_config])
    ]

    supervise(children, strategy: :one_for_all)
  end
end
```

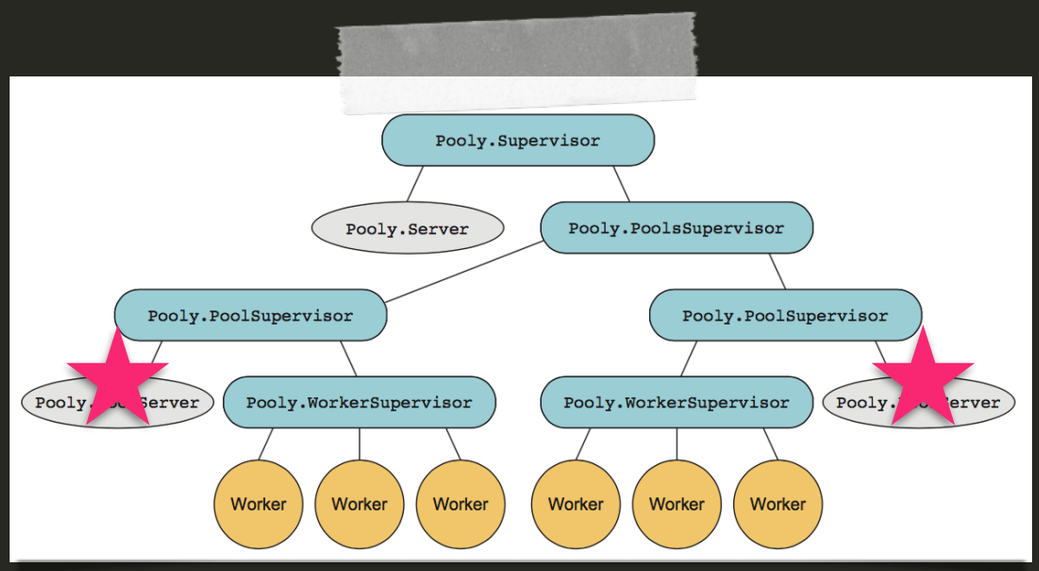


```elixir
defmodule Pooly.PoolSupervisor do
  use Supervisor

  def start_link(pool_config) do
    Supervisor.start_link(__MODULE__, pool_config,
                          name: :"#{pool_config[:name]}Supervisor")
  end

  def init(pool_config) do
    children = [
      worker(Pooly.PoolServer, [self, pool_config])
    ]

    supervise(children, strategy: :one_for_all)
  end
end
```



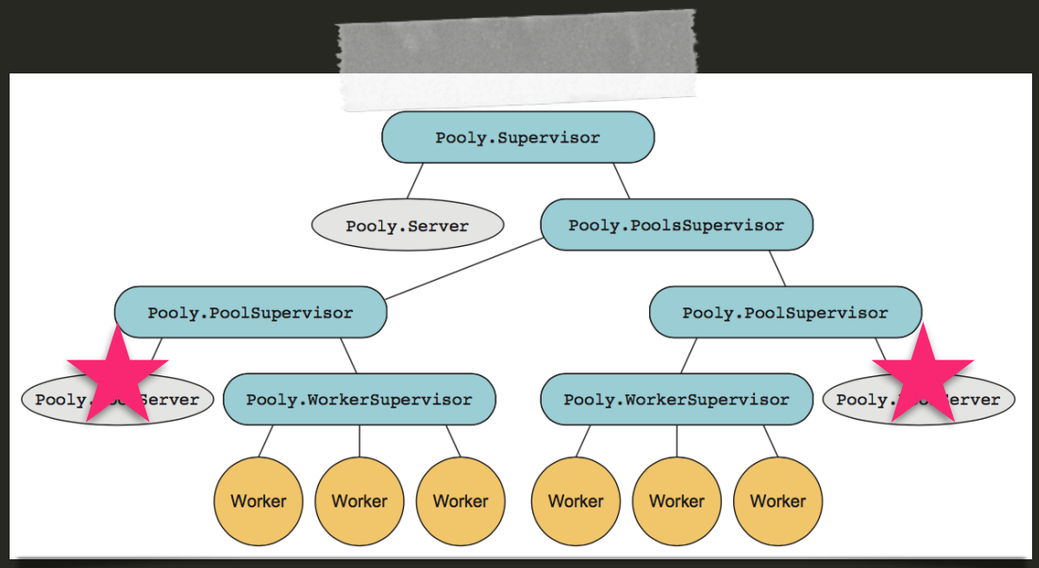
IMPLEMENTING THE POOL SERVER

```
defmodule Pooly.PoolServer do
  defmodule State do
    defstruct
      pool_sup: nil,
      worker_sup: nil,
      monitors: nil,
      size: nil,
      workers: nil,
      name: nil,
      mfa: nil
  end
end
```



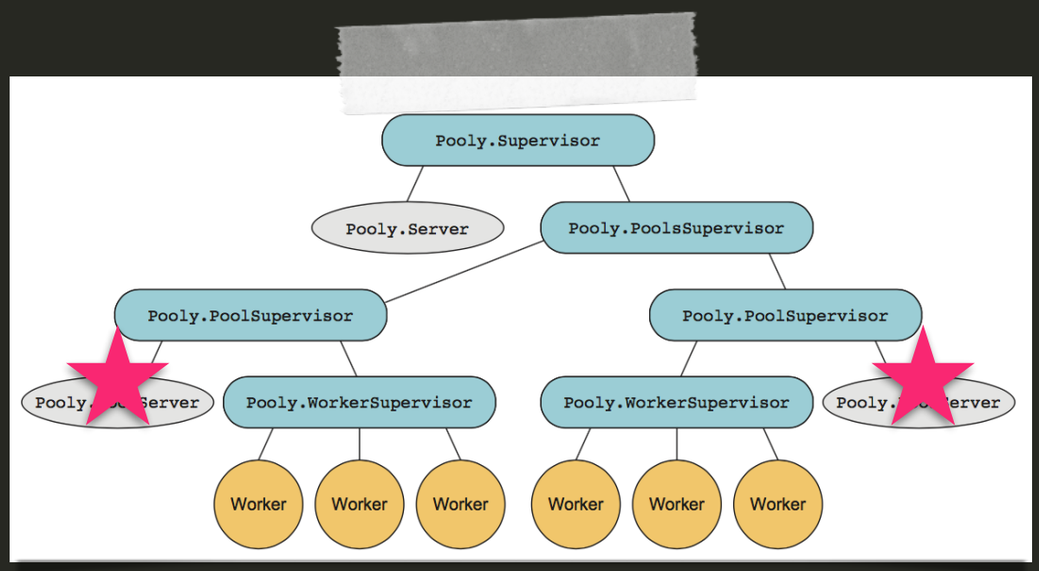
IMPLEMENTING THE POOL SERVER

```
defmodule Pooly.PoolServer do
  defmodule State do
    defstruct
      pool_sup: nil,
      worker_sup: nil,
      monitors: nil,
      size: nil,
      workers: nil,
      name: nil,
      mfa: nil
  end
end
```

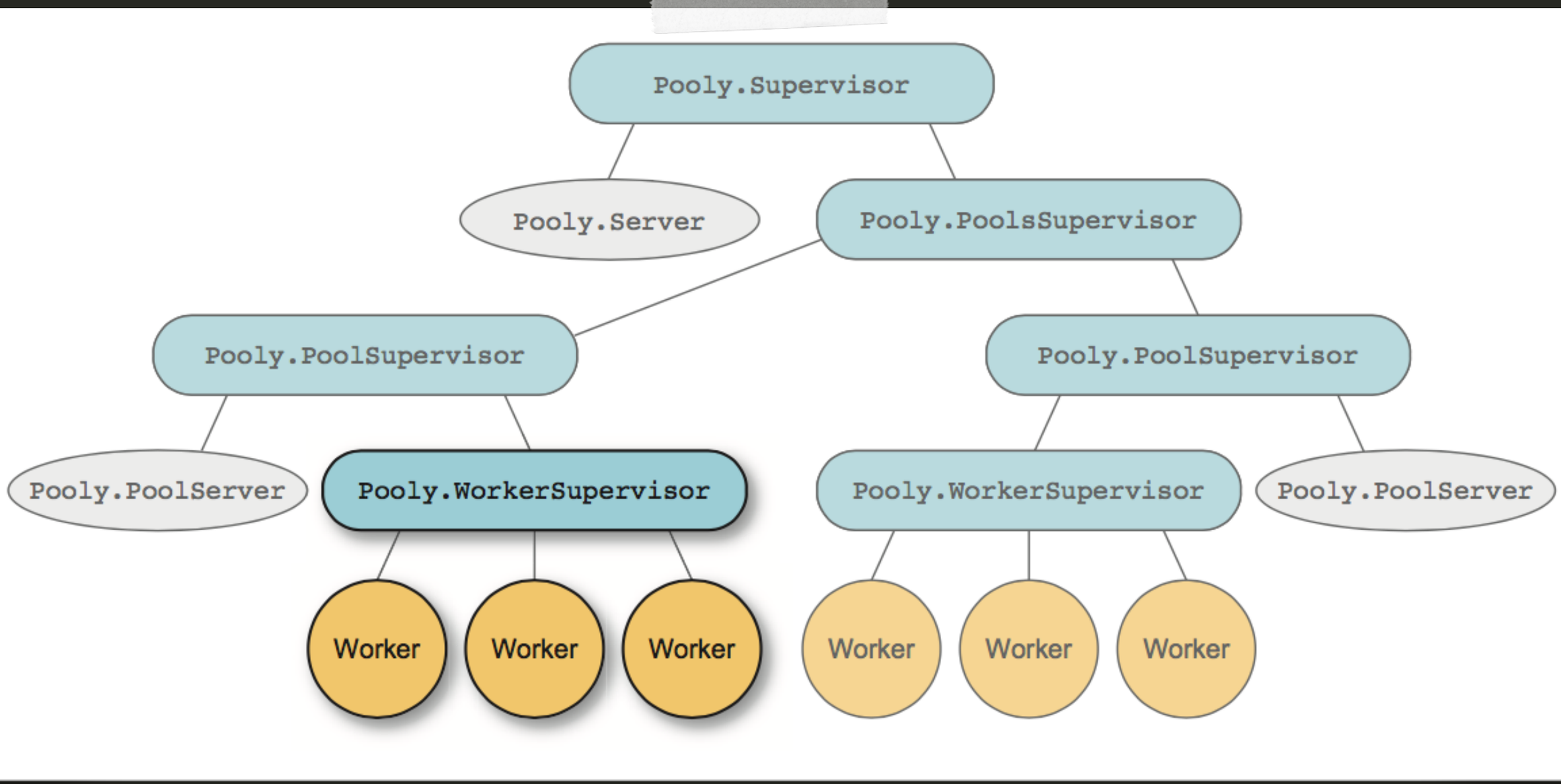


IMPLEMENTING THE POOL SERVER

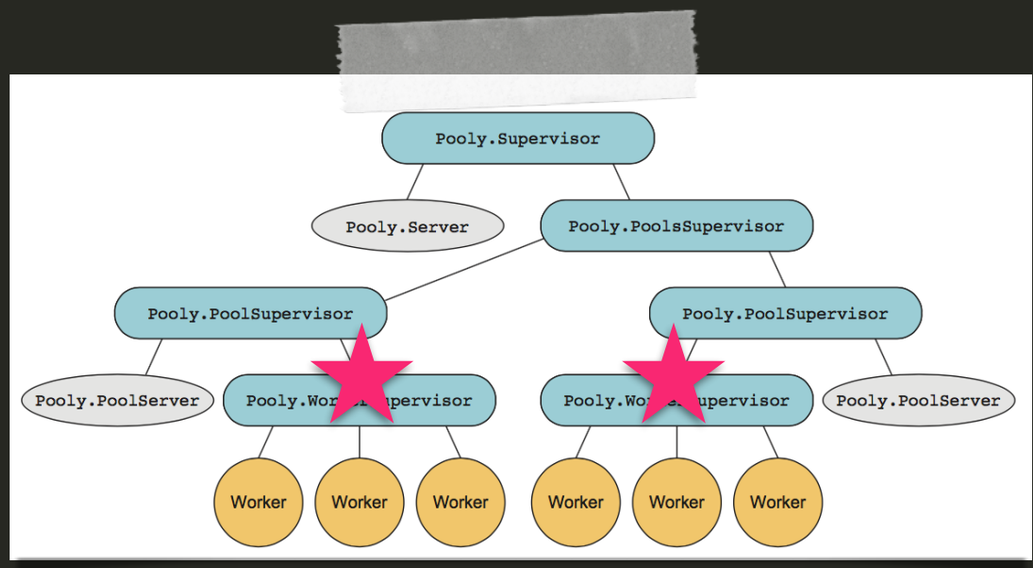
```
defmodule Pooly.PoolServer do
  defmodule State do
    defstruct
      pool_sup: nil,
      worker_sup: nil,
      monitors: nil,
      size: nil,
      workers: nil,
      name: nil,
      mfa: nil
  end
end
```



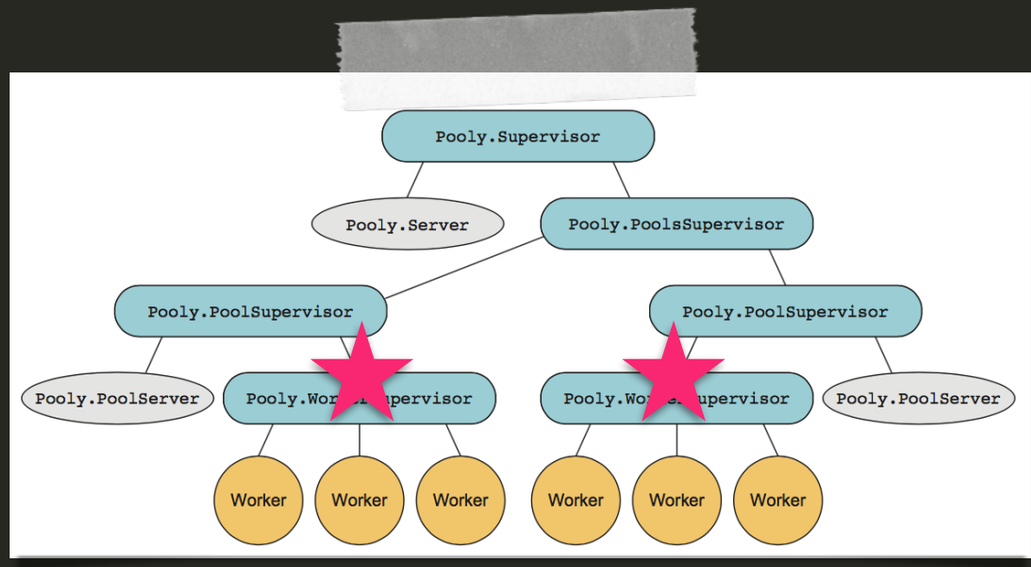
WORKER SUPERVISOR FOR THE POOL



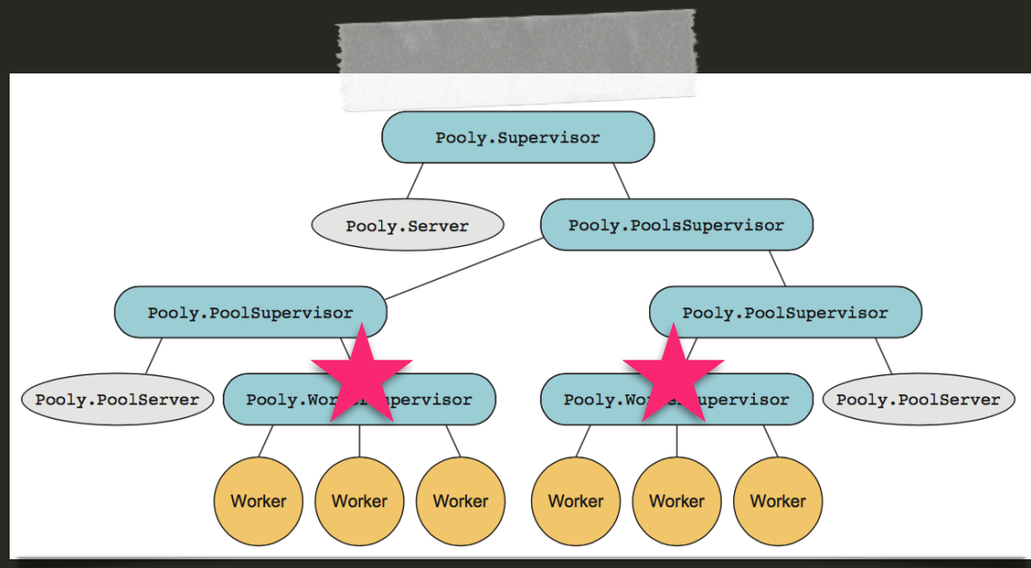
```elixir
defmodule Pooly.WorkerSupervisor do
  use Supervisor

  def start_link(pool_server, {_,_,_} = mfa) do
    Supervisor.start_link(__MODULE__, [pool_server, mfa])
  end

  def init([pool_server, {m,f,a}]) do
    Process.link(pool_server)
    worker_opts = [restart:  :temporary,
                   shutdown: 5000,
                   function: f]

    children = [worker(m, a, worker_opts)]
    opts     = [strategy:     :simple_one_for_one,
                max_restarts: 5,
                max_seconds:  5]
    supervise(children, opts)
  end
end
```



```elixir
defmodule Pooly.WorkerSupervisor do
  use Supervisor

  def start_link(pool_server, {_,_,_} = mfa) do
    Supervisor.start_link(__MODULE__, [pool_server, mfa])
  end

  def init([pool_server, {m,f,a}]) do
    Process.link(pool_server)
    worker_opts = [restart:  :temporary,
                   shutdown: 5000,
                   function: f]

    children = [worker(m, a, worker_opts)]
    opts     = [strategy:     :simple_one_for_one,
                max_restarts: 5,
                max_seconds:  5]
    supervise(children, opts)
  end
end
```

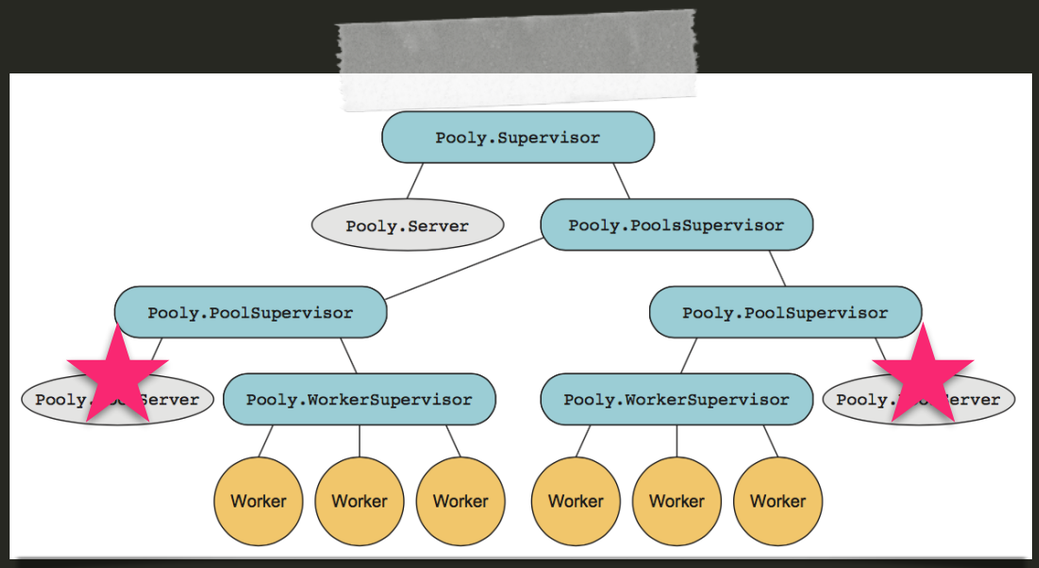


```elixir
defmodule Pooly.WorkerSupervisor do
  use Supervisor

  def start_link(pool_server, {_,_,_} = mfa) do
    Supervisor.start_link(__MODULE__, [pool_server, mfa])
  end

  def init([pool_server, {m,f,a}]) do
    Process.link(pool_server)
    worker_opts = [restart:  :temporary,
                   shutdown: 5000,
                   function: f]

    children = [worker(m, a, worker_opts)]
    opts     = [strategy:     :simple_one_for_one,
                max_restarts: 5,
                max_seconds:  5]
    supervise(children, opts)
  end
end
```



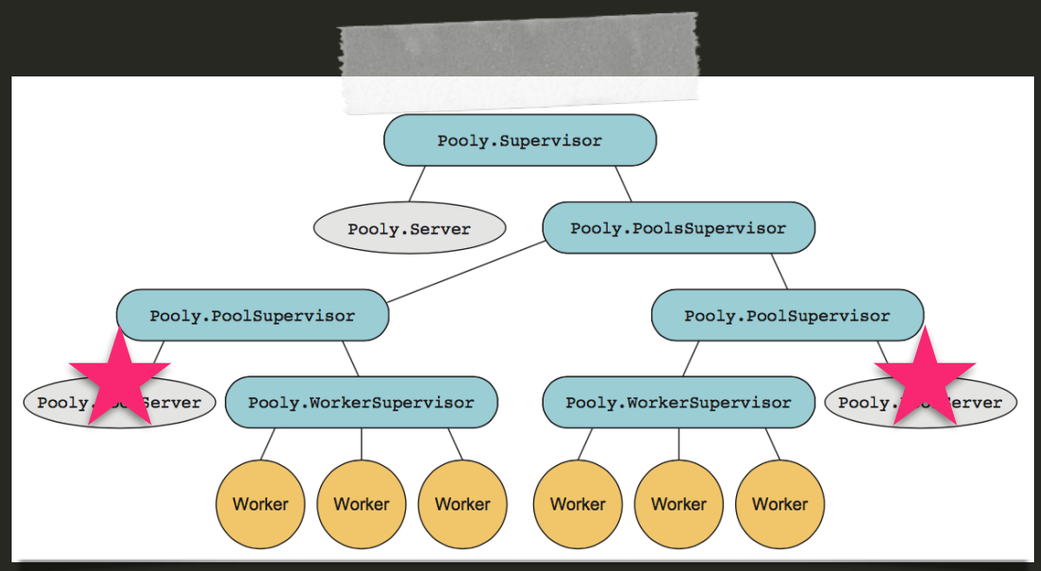
HANDLING A CRASH WHEN WORKER SUPERVISOR GOES DOWN.

```elixir
defmodule Pooly.PoolServer do

  def handle_info({:EXIT, worker_sup, reason}, state) do
    %{worker_sup: ^worker_sup} = state
    {:stop, reason, state}
  end

end
```



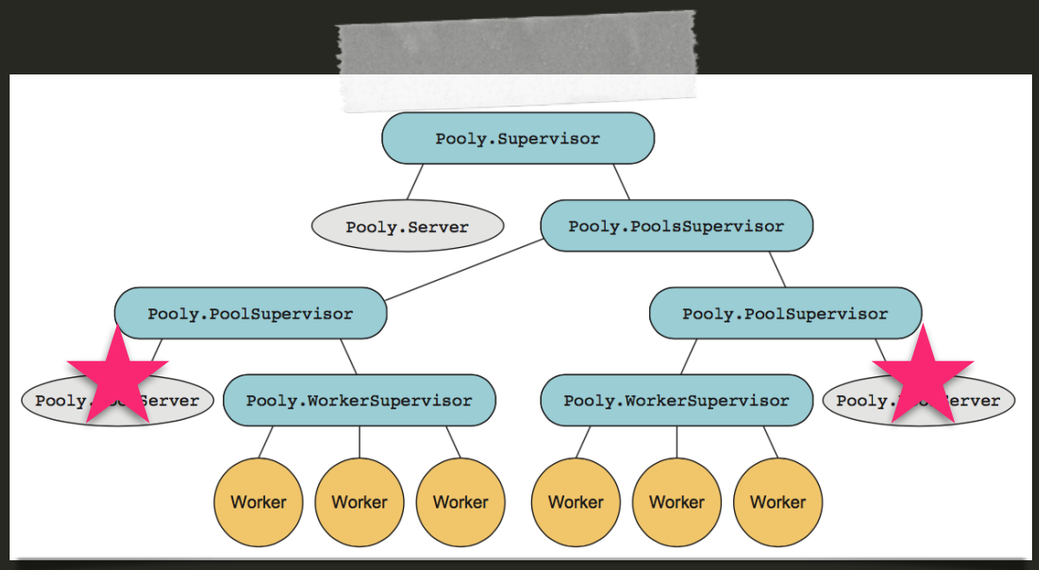
HANDLING A CRASH WHEN WORKER SUPERVISOR GOES DOWN.

```elixir
defmodule Pooly.PoolServer do

  def handle_info({:EXIT, worker_sup, reason}, state) do
    %{worker_sup: ^worker_sup} = state
    {:stop, reason, state}
  end

end
```



HANDLING A CRASH WHEN WORKER SUPERVISOR GOES DOWN.

```
defmodule Pooly.PoolServer do

  def handle_info({:EXIT, worker_sup, reason}, state) do
    %{worker_sup: ^worker_sup} = state
    {:stop, reason, state}
  end

end
```



VERSION 3

TYPE OF POOL	Single	Multiple
CREATION OF WORKERS	Fixed	Dynamic
CONSUMER RECOVERY	No	Yes
WORKER RECOVERY	No	Yes
QUEUEING FOR BUSY WORKERS	No	Yes

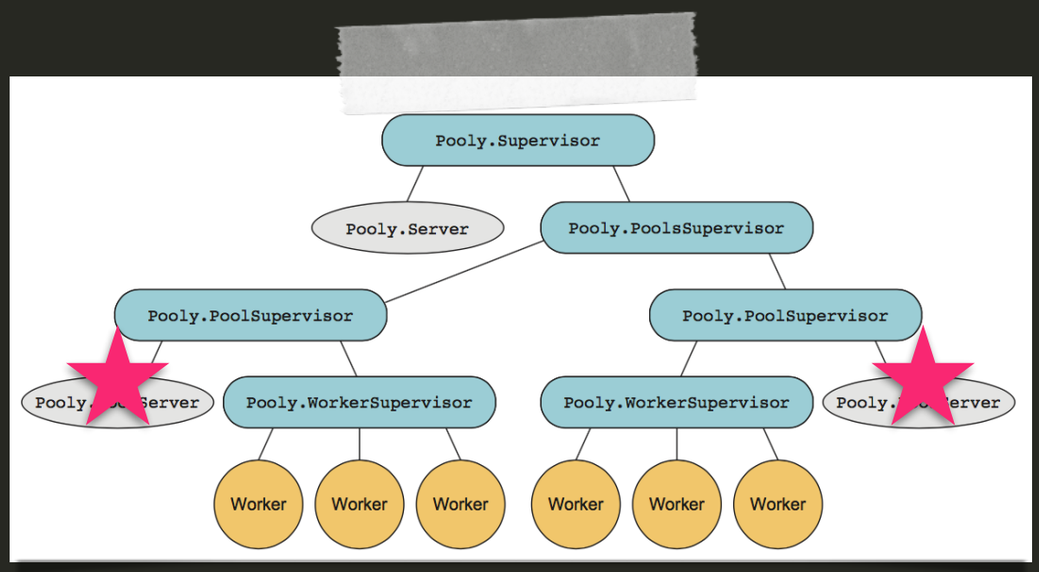
VERSION 4

TYPE OF POOL	Single	Multiple
CREATION OF WORKERS	Fixed	Dynamic
CONSUMER RECOVERY	No	Yes
WORKER RECOVERY	No	Yes
QUEUEING FOR BUSY WORKERS	No	Yes

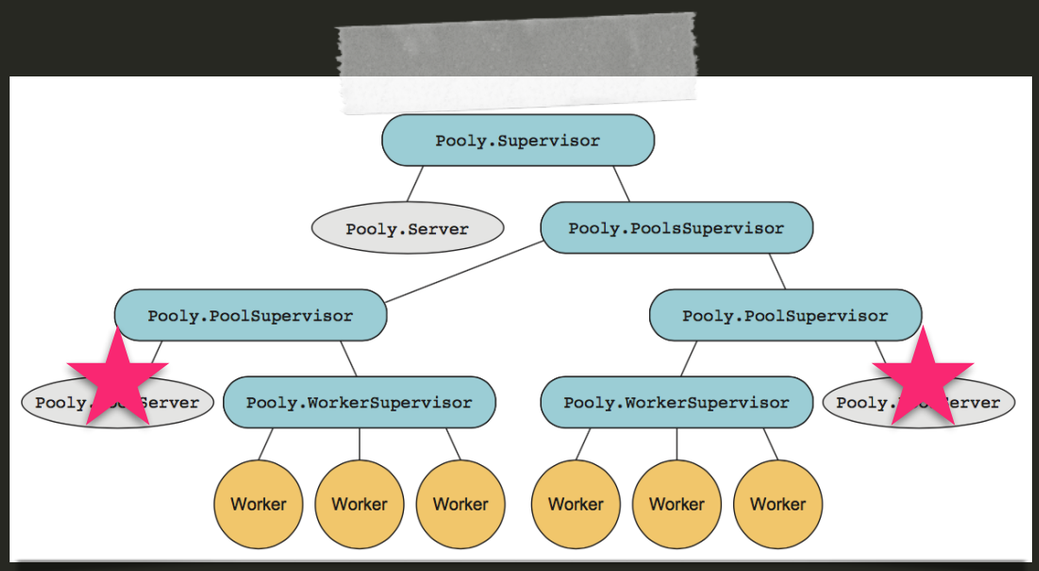
IMPLEMENTING OVERFLOW OVERFLOW OVERFLOW

```elixir
defmodule Pooly do
  def start(_type, _args) do
    pools_config =
      [
        [name: "ChuckNorris",
         mfa: {ChuckFetcher, :start_link, []},
         size: 2,
         max_overflow: 3
        ],
        [name: "StarWars",
         mfa: {SwapiFetcher, :start_link, []},
         size: 4,
         max_overflow: 3
        ]
      ]
    start_pools(pools_config)
  end
end
```

```elixir
defmodule Pooly.PoolServer do
  defmodule State do
    defstruct
      pool_sup: nil,
      worker_sup: nil,
      monitors: nil,
      size: nil,
      workers: nil,
      name: nil,
      mfa: nil,
      overflow: nil,
      max_overflow: nil
    end
  end
end
```

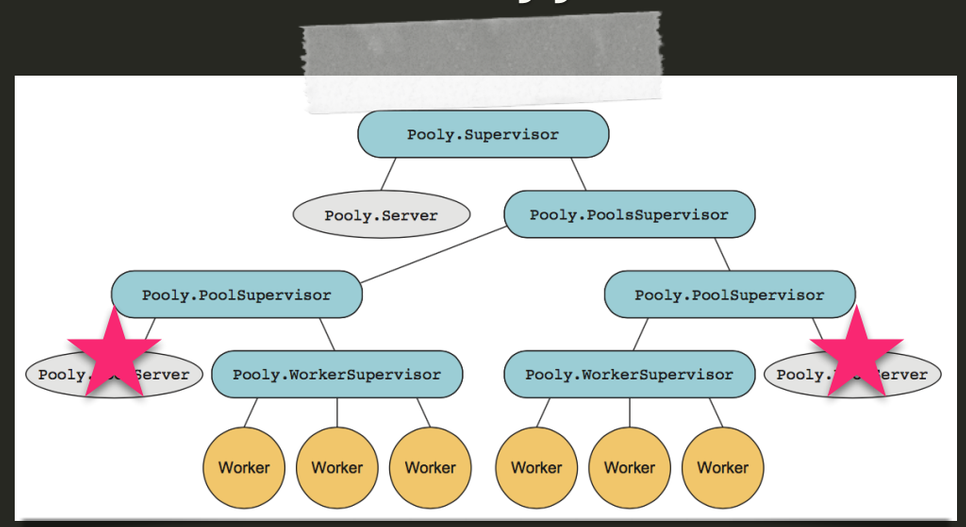


```elixir
defmodule Pooly.PoolServer do
  defmodule State do
    defstruct
      pool_sup: nil,
      worker_sup: nil,
      monitors: nil,
      size: nil,
      workers: nil,
      name: nil,
      mfa: nil,
      overflow: nil,
      max_overflow: nil
    end
  end
end
```



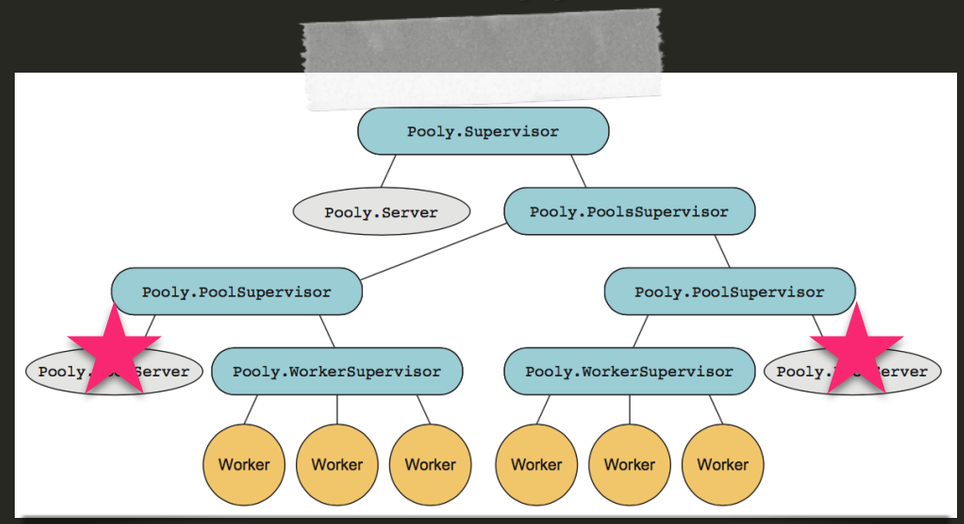
OVERFLOW: HANDLING WORKER CHECKOUTS

```
defmodule Pooly.PoolServer do
  def handle_call(:checkout, {from_pid, _ref} = from, state) do
    %{worker_sup:   worker_sup,
      workers:      workers,
      monitors:     monitors,
      overflow:     overflow,
      max_overflow: max_overflow} = state

    case workers do
      [worker|rest] ->
        # ...
        {:reply, worker, %{state | workers: rest}}
      [] when max_overflow > 0 and overflow < max_overflow ->
        {worker, ref} = new_worker(worker_sup, from_pid)
        true = :ets.insert(monitors, {worker, ref})
        {:reply, worker, %{state | overflow: overflow+1}}
      [] ->
        {:reply, :full, state};
    end
  end
end
```



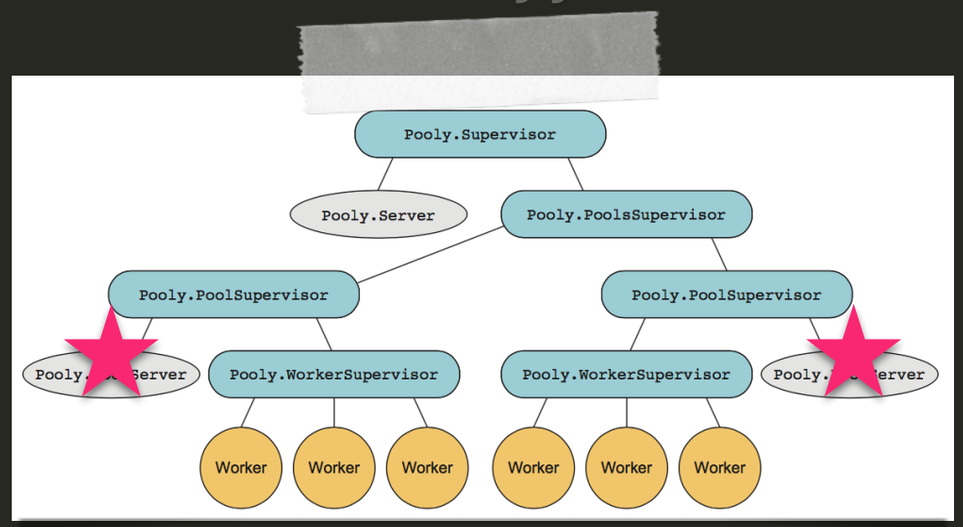
OVERFLOW: HANDLING WORKER CHECKOUTS

```
defmodule Pooly.PoolServer do
  def handle_call(:checkout, {from_pid, _ref} = from, state) do
    %{worker_sup:   worker_sup,
      workers:      workers,
      monitors:     monitors,
      overflow:     overflow,
      max_overflow: max_overflow} = state

    case workers do
      [worker|rest] ->
        # ...
        {:reply, worker, %{state | workers: rest}}
      [] when max_overflow > 0 and overflow < max_overflow ->
        {worker, ref} = new_worker(worker_sup, from_pid)
        true = :ets.insert(monitors, {worker, ref})
        {:reply, worker, %{state | overflow: overflow+1}}
      [] ->
        {:reply, :full, state};
    end
  end
end
```



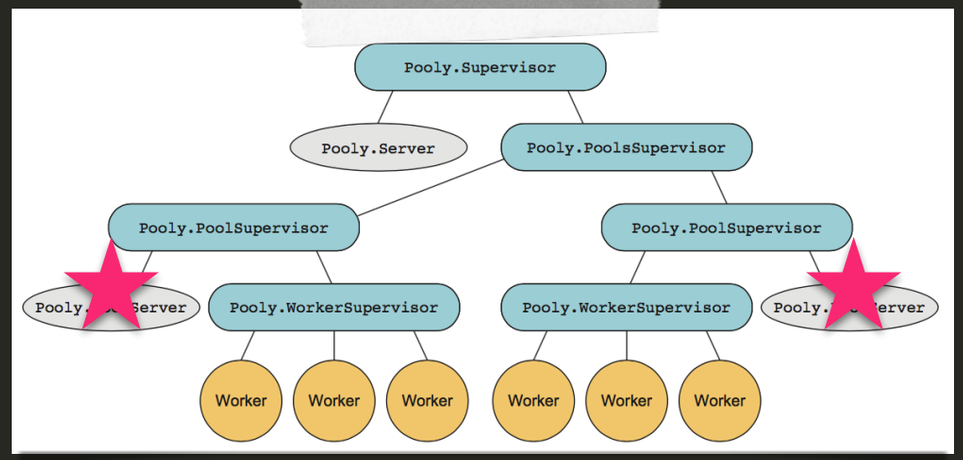
OVERFLOW: HANDLING WORKER CHECKOUTS

```
defmodule Pooly.PoolServer do
  def handle_call(:checkout, {from_pid, _ref} = from, state) do
    %{worker_sup:   worker_sup,
      workers:      workers,
      monitors:     monitors,
      overflow:     overflow,
      max_overflow: max_overflow} = state

    case workers do
      [worker|rest] ->
        # ...
        {:reply, worker, %{state | workers: rest}}
      [] when max_overflow > 0 and overflow < max_overflow ->
        {worker, ref} = new_worker(worker_sup, from_pid)
        true = :ets.insert(monitors, {worker, ref})
        {:reply, worker, %{state | overflow: overflow+1}}
      [] ->
        {:reply, :full, state};
    end
  end
end
```



OVERFLOW: HANDLING WORKER CHECKOUTS

```elixir
defmodule Pooly.PoolServer do
  def handle_call(:checkout, {from_pid, _ref} = from, state) do
    %{worker_sup:   worker_sup,
      workers:      workers,
      monitors:     monitors,
      overflow:     overflow,
      max_overflow: max_overflow} = state

    case workers do
      [worker|rest] ->
        # ...
        {:reply, worker, %{state | workers: rest}}
      [] when max_overflow > 0 and overflow < max_overflow ->
        {worker, ref} = new_worker(worker_sup, from_pid)
        true = :ets.insert(monitors, {worker, ref})
        {:reply, worker, %{state | overflow: overflow+1}}
      [] ->
        {:reply, :full, state};
    end
  end
end
```



OVERFLOW: HANDLING WORKER CHECKINS

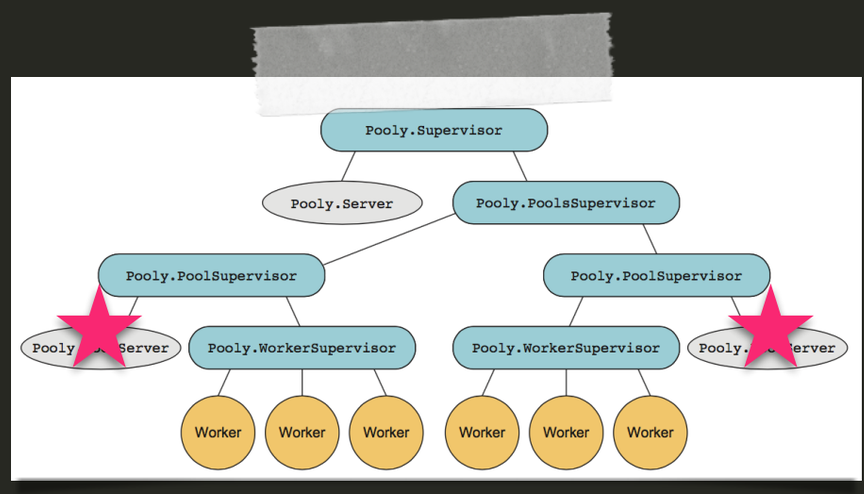
Previously:

```
{:noreply, %{state | workers: [pid|workers]}}
```

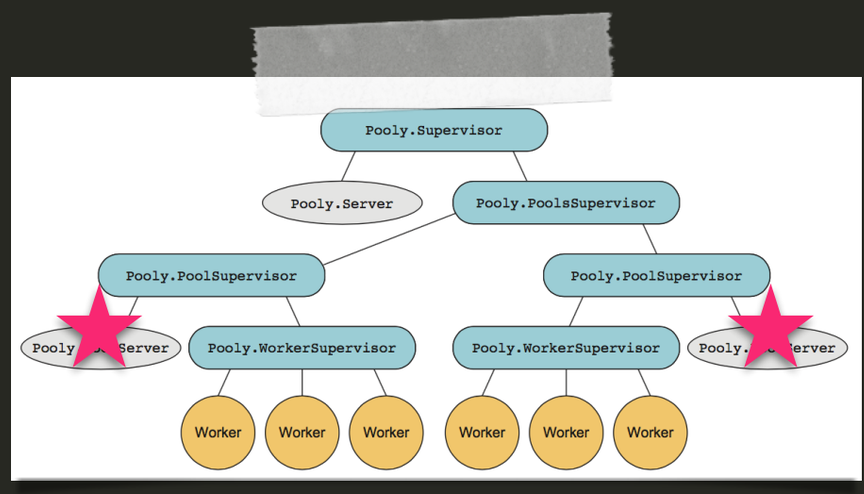
Now:

Worker dismissal: **UNLINK** + **TERMINATE CHILD**

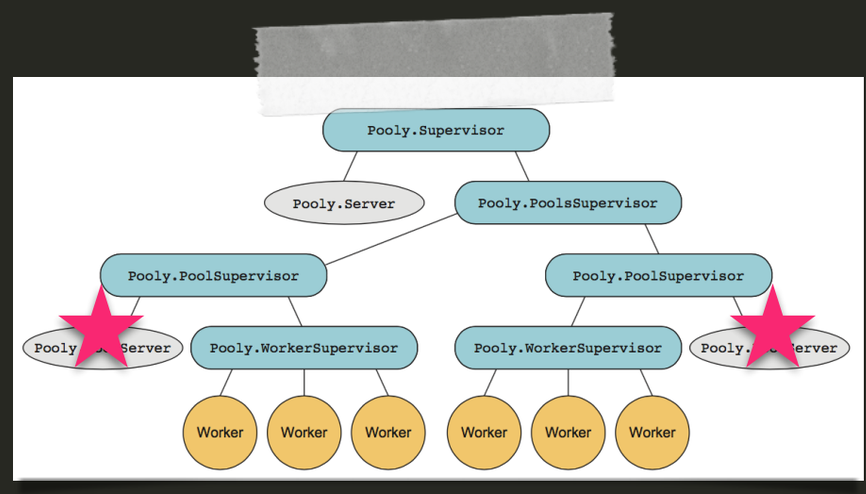
```elixir
defp handle_checkin(pid, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow} = state

  if overflow > 0 do
    :ok = dismiss_worker(worker_sup, pid)
    %{state | waiting: empty, overflow: overflow-1}
  else
    %{state | waiting: empty,
              workers: [pid|workers], overflow: 0}
  end
end

defp dismiss_worker(sup, pid) do
  true = Process.unlink(pid)
  Supervisor.terminate_child(sup, pid)
end
```



```elixir
defp handle_checkin(pid, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow} = state

  if overflow > 0 do
    :ok = dismiss_worker(worker_sup, pid)
    %{state | waiting: empty, overflow: overflow-1}
  else
    %{state | waiting: empty,
              workers: [pid|workers], overflow: 0}
  end
end

defp dismiss_worker(sup, pid) do
  true = Process.unlink(pid)
  Supervisor.terminate_child(sup, pid)
end
```



```elixir
defp handle_checkin(pid, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow} = state

  if overflow > 0 do
    :ok = dismiss_worker(worker_sup, pid)
    %{state | waiting: empty, overflow: overflow-1}
  else
    %{state | waiting: empty,
              workers: [pid|workers], overflow: 0}
  end
end

defp dismiss_worker(sup, pid) do
  true = Process.unlink(pid)
  Supervisor.terminate_child(sup, pid)
end
```

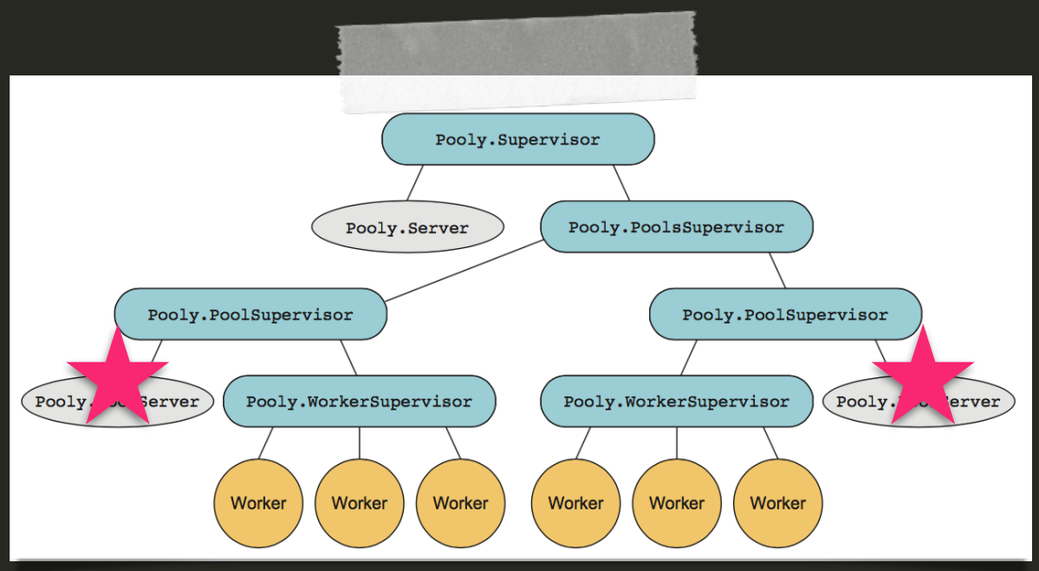


Handling Worker

HANDLING WORKER EXITS

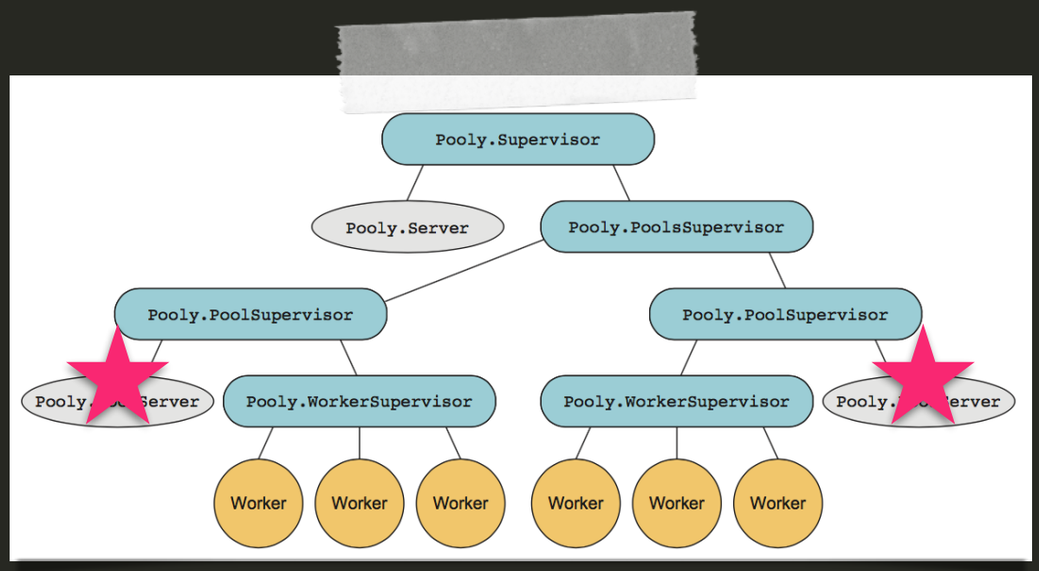
```
defmodule Pooly.PoolServer do

  defp handle_worker_exit(pid, state) do
    %{worker_sup: worker_sup,
      workers: workers,
      monitors: monitors,
      overflow: overflow} = state
    if overflow > 0 do
      %{state | overflow: overflow-1}
    else
      %{state | workers: [new_worker(worker_sup)|workers]}
    end
  end

end
```

```
defp handle_checkin(pid, state) do
  if overflow > 0 do
    :ok = dismiss_worker(worker_sup,
pid)
    %{state | waiting: empty,
              overflow: overflow-1}
  else
    %{state | waiting: empty,
              workers: [pid|workers],
              overflow: 0}
  end
end
```



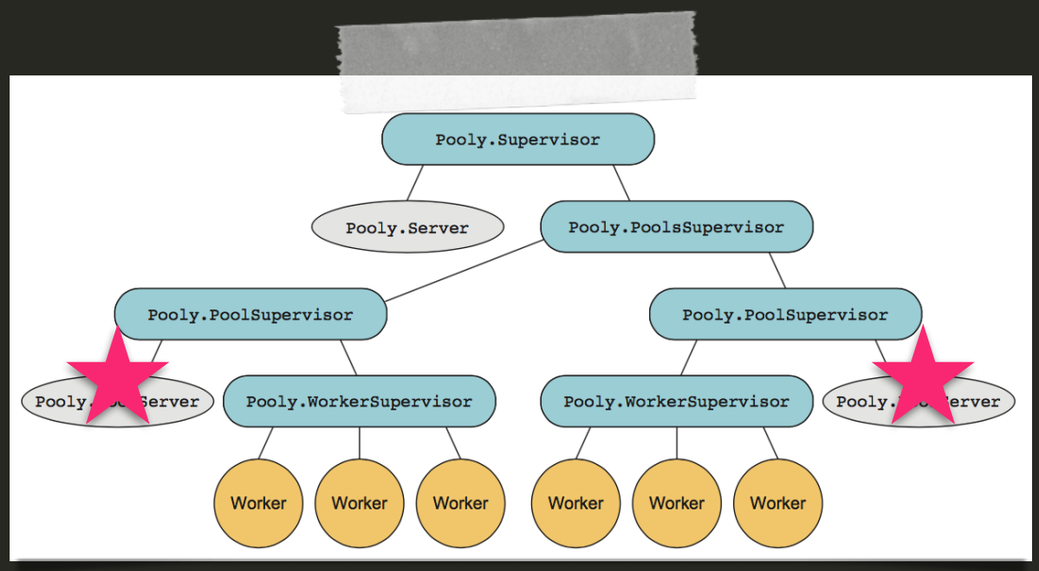
HANDLING WORKER EXITS

```
defmodule Pooly.PoolServer do

  defp handle_worker_exit(pid, state) do
    %{worker_sup: worker_sup,
      workers: workers,
      monitors: monitors,
      overflow: overflow} = state
    if overflow > 0 do
      %{state | overflow: overflow-1}
    else
      %{state | workers: [new_worker(worker_sup)|workers]}
    end
  end

end
```



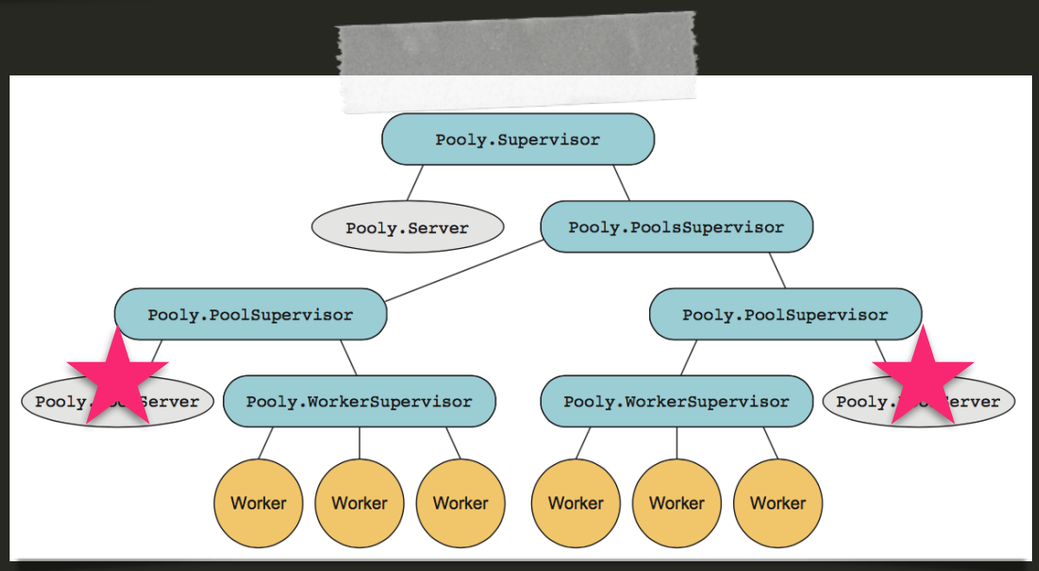
HANDLING WORKER EXITS

```
defmodule Pooly.PoolServer do

  defp handle_worker_exit(pid, state) do
    %{worker_sup: worker_sup,
      workers: workers,
      monitors: monitors,
      overflow: overflow} = state
    if overflow > 0 do
      %{state | overflow: overflow-1}
    else
      %{state | workers: [new_worker(worker_sup)|workers]}
    end
  end

end
```



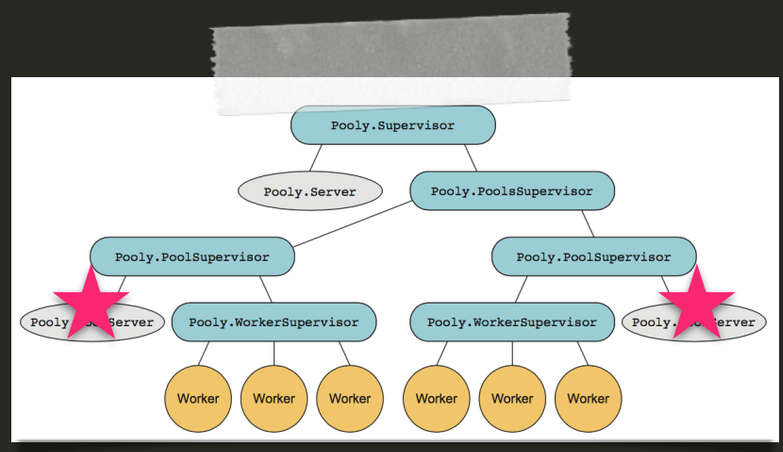
HANDLING WORKER EXITS

```
defmodule Pooly.PoolServer do

  def handle_info({:EXIT, pid, _reason}, state) do
    %{monitors: monitors,
      workers: workers,
      worker_sup: worker_sup} = state

    case :ets.lookup(monitors, pid) do
      [{pid, ref}] ->
        #
        new_state = handle_worker_exit(pid, state)
        {:noreply, new_state}
      _ ->
        {:noreply, state}
    end
  end

end
```

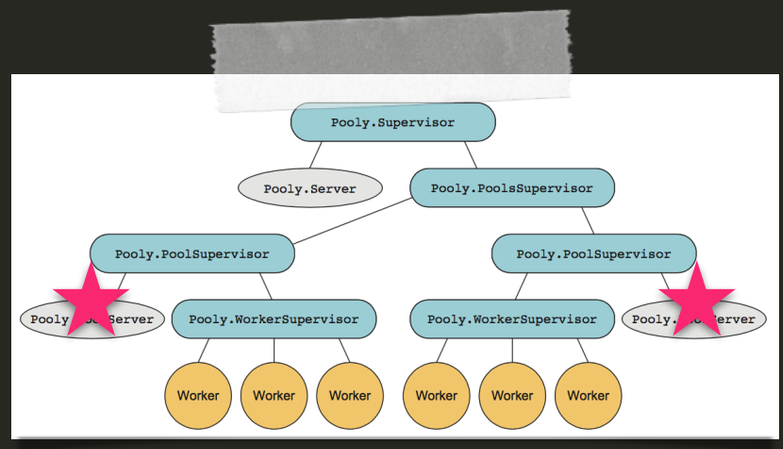


WHAT IF A CONSUMER PROCESS IS WILLING TO BLOCK?

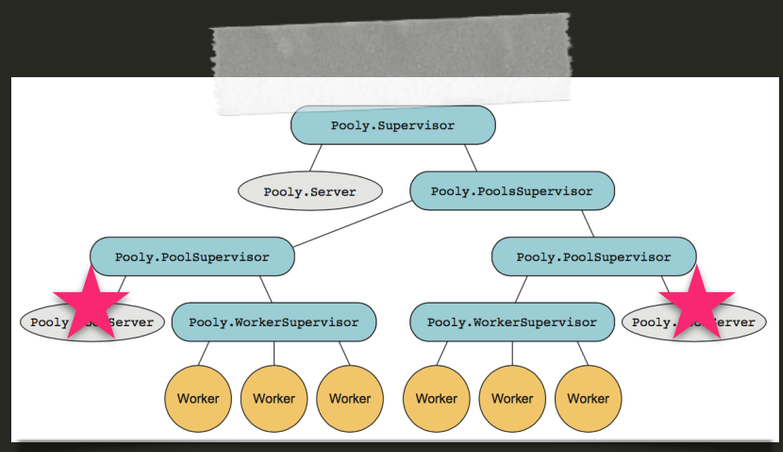
```elixir
defmodule Pooly.PoolServer do
  defmodule State do
    defstruct
    ...,
    waiting: nil,
    overflow: nil,
    max_overflow: nil
  end

  def init([pool_sup, pool_config]) when is_pid(pool_sup) do
    Process.flag(:trap_exit, true)
    monitors = :ets.new(:monitors, [:private])
    waiting  = :queue.new
    state    = %State{pool_sup: pool_sup,
                      monitors: monitors,
                      waiting: waiting,
                      overflow: 0}

    init(pool_config, state)
  end
end
```



```elixir
defmodule Pooly.PoolServer do
  defmodule State do
    defstruct
    ...,
    waiting: nil,
    overflow: nil,
    max_overflow: nil
  end

  def init([pool_sup, pool_config]) when is_pid(pool_sup) do
    Process.flag(:trap_exit, true)
    monitors = :ets.new(:monitors, [:private])
    waiting  = :queue.new
    state    = %State{pool_sup: pool_sup,
                      monitors: monitors,
                      waiting: waiting,
                      overflow: 0}

    init(pool_config, state)
  end
end
```

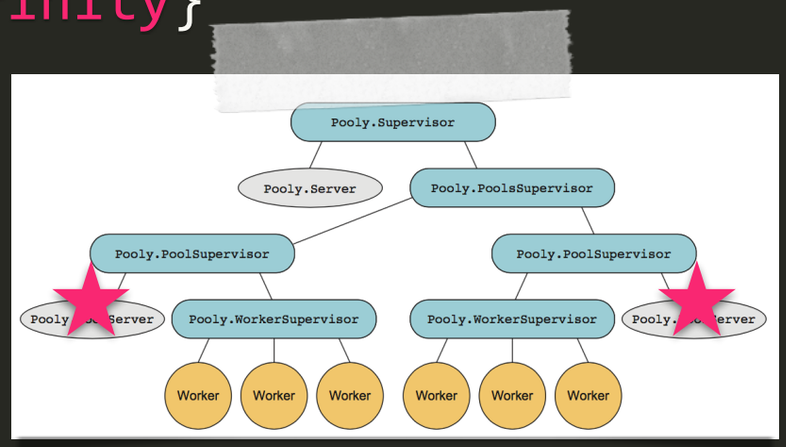


```elixir
defmodule Pooly.PoolServer do
  defmodule State do
    defstruct
    ...,
    waiting: nil,
    overflow: nil,
    max_overflow: nil
  end

  def init([pool_sup, pool_config]) when is_pid(pool_sup) do
    Process.flag(:trap_exit, true)
    monitors = :ets.new(:monitors, [:private])
    waiting  = :queue.new
    state    = %State{pool_sup: pool_sup,
                      monitors: monitors,
                      waiting: waiting,
                      overflow: 0}

    init(pool_config, state)
  end
end
```



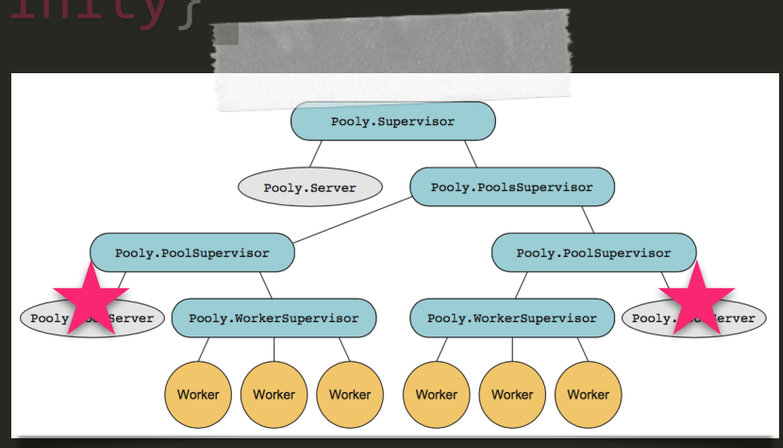
QUEUING: CHECKING OUT

```
def handle_call({:checkout, block}, {from_pid, _ref} = from, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow,
    max_overflow: max_overflow} = state

  case workers do
    [worker|rest] ->
      # ...
    [] when max_overflow > 0 and overflow < max_overflow ->
      # …

    [] when block == true ->
      ref = Process.monitor(from_pid)
      waiting = :queue.in({from, ref}, waiting)
      {:noreply, %{state | waiting: waiting}, :infinity}

    [] ->
      {:reply, :full, state}
  end
end
```



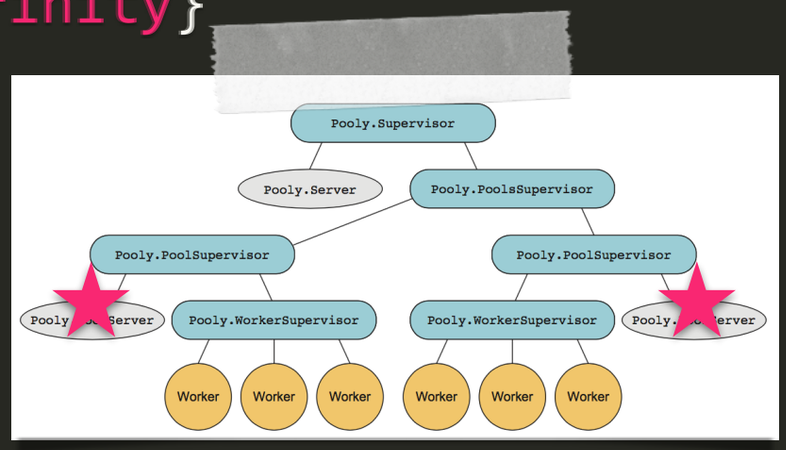
QUEUING: CHECKING OUT

```
def handle_call({:checkout, block}, {from_pid, _ref} = from, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow,
    max_overflow: max_overflow} = state

  case workers do
    [worker|rest] ->
      # …

    [] when max_overflow > 0 and overflow < max_overflow ->
      # …

    [] when block == true ->
      ref = Process.monitor(from_pid)
      waiting = :queue.in({from, ref}, waiting)
      {:noreply, %{state | waiting: waiting}, :infinity}

    [] ->
      {:reply, :full, state}
  end
end
```



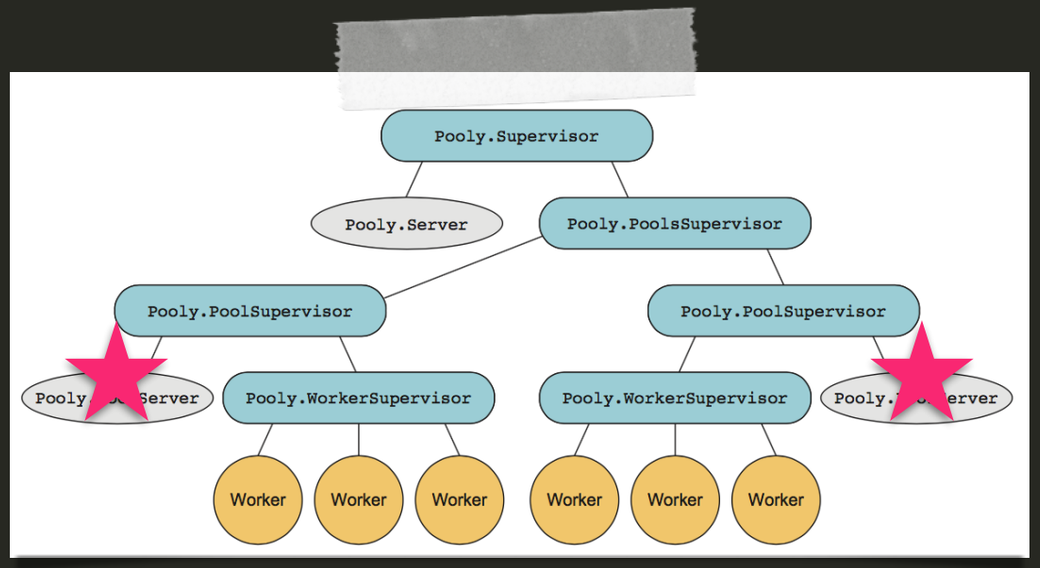
QUEUING: CHECKING OUT

```
def handle_call({:checkout, block}, {from_pid, _ref} = from, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow,
    max_overflow: max_overflow} = state

  case workers do
    [worker|rest] ->
      # …

    [] when max_overflow > 0 and overflow < max_overflow ->
      # …

    [] when block == true ->
      ref = Process.monitor(from_pid)
      waiting = :queue.in({from, ref}, waiting)
      {:noreply, %{state | waiting: waiting}, :infinity}

    [] ->
      {:reply, :full, state}
  end
end
```



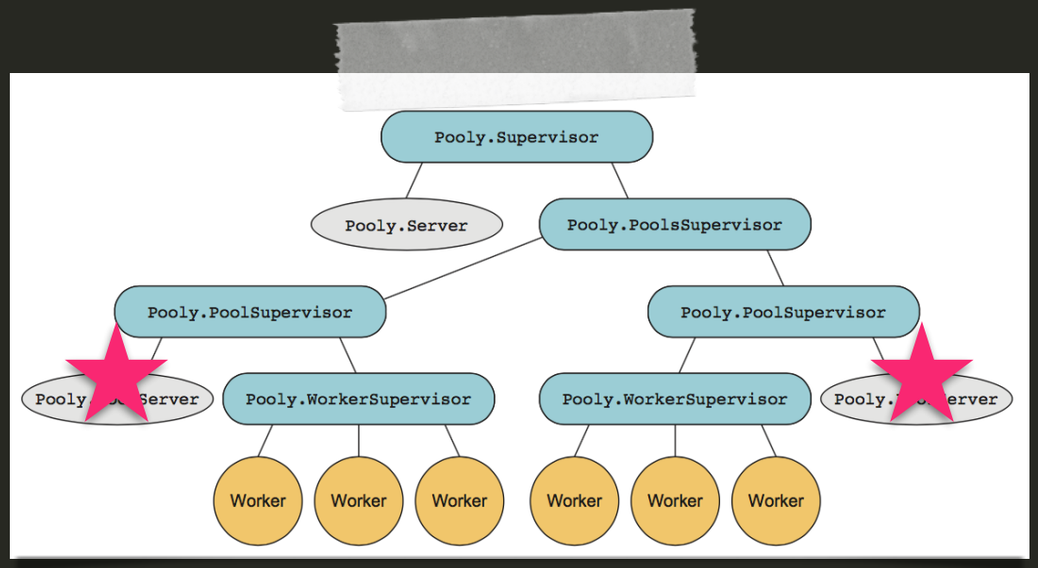
QUEUING: CHECKING IN

```
def handle_checkin(pid, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow} = state

  case :queue.out(waiting) do
    {{:value, {from, ref}}, left} ->
      true = :ets.insert(monitors, {pid, ref})
      GenServer.reply(from, pid)
      %{state | waiting: left}

    {:empty, empty} when overflow > 0 ->
      :ok = dismiss_worker(worker_sup, pid)
      %{state | waiting: empty, overflow: overflow-1}

    {:empty, empty} ->
      %{state | waiting: empty, workers: [pid|workers], overflow: 0}
  end
end
```



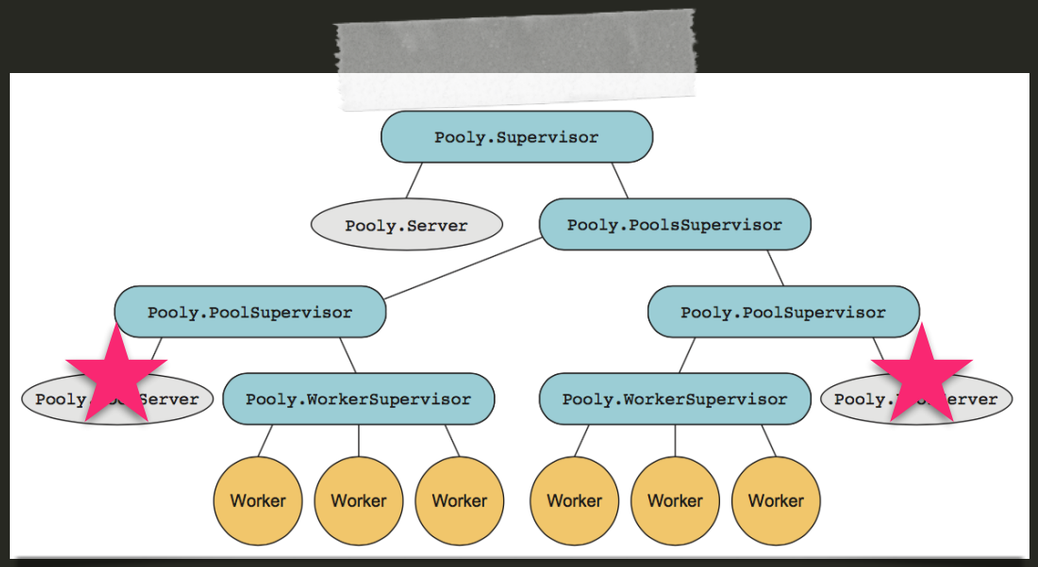
QUEUING: CHECKING IN

```elixir
def handle_checkin(pid, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow} = state

  case :queue.out(waiting) do
    {{:value, {from, ref}}, left} ->
      true = :ets.insert(monitors, {pid, ref})
      GenServer.reply(from, pid)
      %{state | waiting: left}

    {:empty, empty} when overflow > 0 ->
      :ok = dismiss_worker(worker_sup, pid)
      %{state | waiting: empty, overflow: overflow-1}

    {:empty, empty} ->
      %{state | waiting: empty, workers: [pid|workers], overflow: 0}
  end
end
```



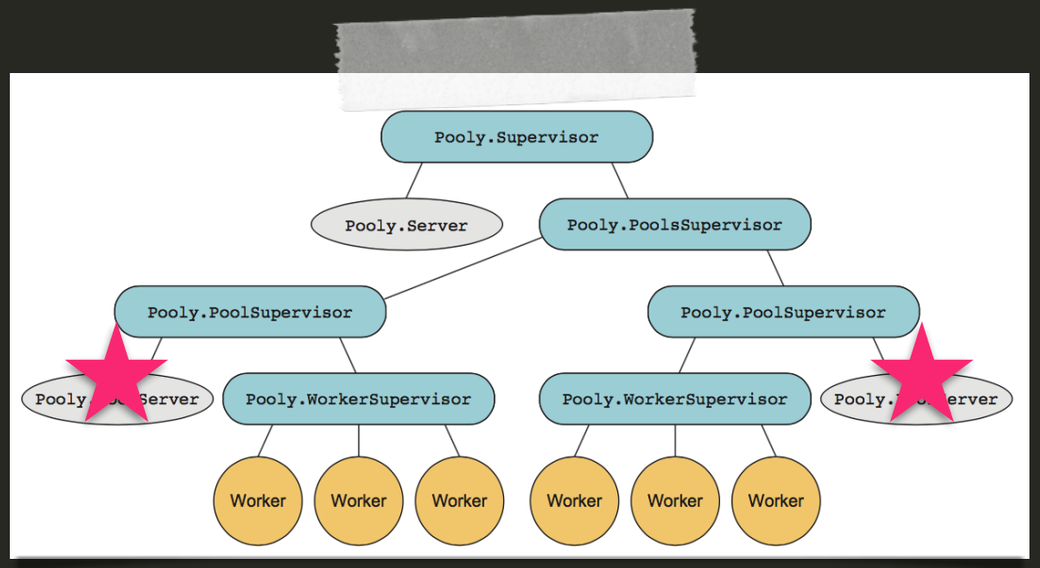
QUEUING: CHECKING IN

```
def handle_checkin(pid, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow} = state

  case :queue.out(waiting) do
    {{:value, {from, ref}}, left} ->
      true = :ets.insert(monitors, {pid, ref})
      GenServer.reply(from, pid)
      %{state | waiting: left}

    {:empty, empty} when overflow > 0 ->
      :ok = dismiss_worker(worker_sup, pid)
      %{state | waiting: empty, overflow: overflow-1}

    {:empty, empty} ->
      %{state | waiting: empty, workers: [pid|workers], overflow: 0}
  end
end
```



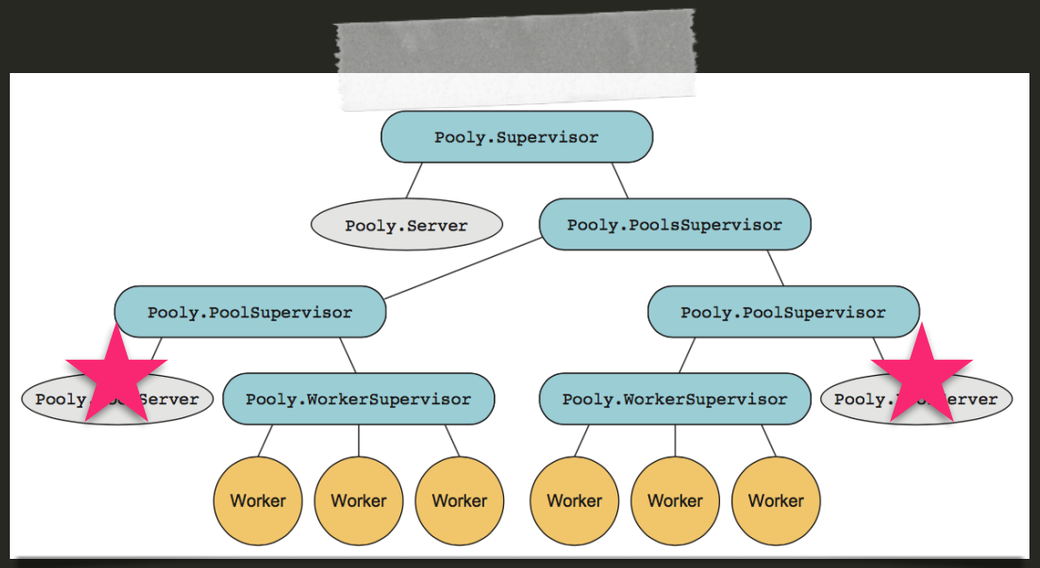
QUEUING: CHECKING IN

```
def handle_checkin(pid, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow} = state

  case :queue.out(waiting) do
    {{:value, {from, ref}}, left} ->
      true = :ets.insert(monitors, {pid, ref})
      GenServer.reply(from, pid)
      %{state | waiting: left}

    {:empty, empty} when overflow > 0 ->
      :ok = dismiss_worker(worker_sup, pid)
      %{state | waiting: empty, overflow: overflow-1}

    {:empty, empty} ->
      %{state | waiting: empty, workers: [pid|workers], overflow: 0}
  end
end
```



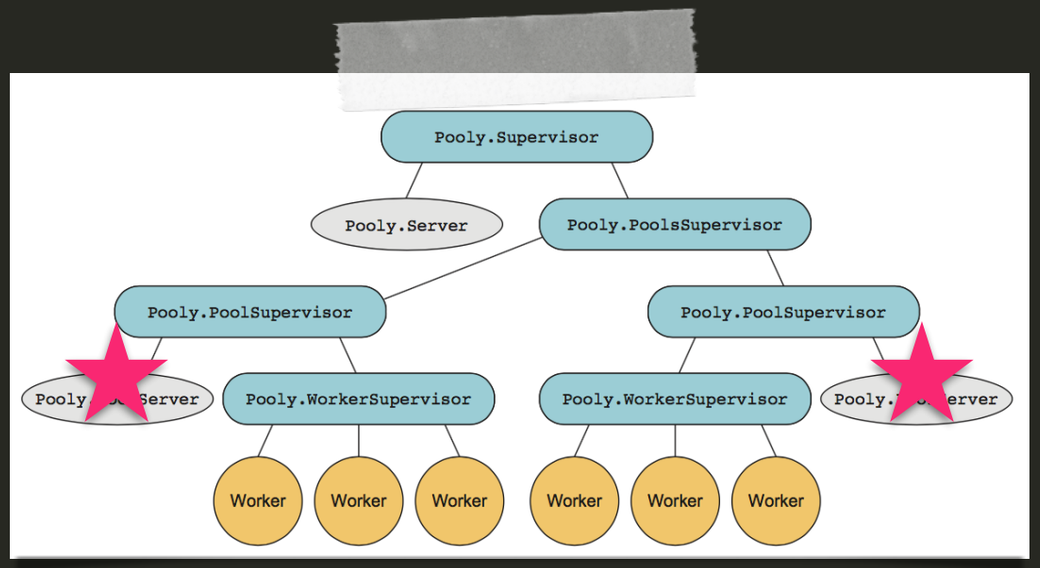
QUEUING: CHECKING IN

```elixir
def handle_checkin(pid, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow} = state

  case :queue.out(waiting) do
    {{:value, {from, ref}}, left} ->
      true = :ets.insert(monitors, {pid, ref})
      GenServer.reply(from, pid)
      %{state | waiting: left}

    {:empty, empty} when overflow > 0 ->
      :ok = dismiss_worker(worker_sup, pid)
      %{state | waiting: empty, overflow: overflow-1}

    {:empty, empty} ->
      %{state | waiting: empty, workers: [pid|workers], overflow: 0}
  end
end
```



QUEUING: CHECKING IN

```elixir
def handle_checkin(pid, state) do
  %{worker_sup:   worker_sup,
    workers:      workers,
    monitors:     monitors,
    waiting:      waiting,
    overflow:     overflow} = state

  case :queue.out(waiting) do
    {{:value, {from, ref}}, left} ->
      true = :ets.insert(monitors, {pid, ref})
      GenServer.reply(from, pid)
      %{state | waiting: left}

    {:empty, empty} when overflow > 0 ->
      :ok = dismiss_worker(worker_sup, pid)
      %{state | waiting: empty, overflow: overflow-1}

    {:empty, empty} ->
      %{state | waiting: empty, workers: [pid|workers], overflow: 0}
  end
end
```



How Do You **ENSURE**

THAT EVERY

CHECK-OUT

IS FOLLOWED BY A

CHECK-IN?

TRANSACTIONS

```
def transaction(pool_name, fun, timeout) do
  worker = checkout(pool_name, true, timeout)
  try do
    fun.(worker)
  after
    checkin(pool_name, worker)
  end
end
```

TRANSACTIONS

```
def transaction(pool_name, fun, timeout) do
  worker = checkout(pool_name, true, timeout)
  try do
    fun.(worker)
  after
    checkin(pool_name, worker)
  end
end
```

TRANSACTIONS

```
def transaction(pool_name, fun, timeout) do
  worker = checkout(pool_name, true, timeout)
  try do
    fun.(worker)
  after
    checkin(pool_name, worker)
  end
end
```

TRANSACTIONS

```
def transaction(pool_name, fun, timeout) do
  worker = checkout(pool_name, true, timeout)
  try do
    fun.(worker)
  after
    checkin(pool_name, worker)
  end
end
```

TRANSACTIONS

```
def transaction(pool_name, fun, timeout) do
  worker = checkout(pool_name, true, timeout)
  try do
    fun.(worker)
  after
    checkin(pool_name, worker)
  end
end
```

TRANSACTIONS

```
def transaction(pool_name, fun, timeout) do
  worker = checkout(pool_name, true, timeout)
  try do
    fun.(worker)
  after
    checkin(pool_name, worker)
  end
end
```

TRANSACTIONS

```
def transaction(pool_name, fun, timeout) do
  worker = checkout(pool_name, true, timeout)
  try do
    fun.(worker)
  after
    checkin(pool_name, worker)
  end
end
```

TRANSACTIONS

```elixir
tasks = 1..5 |> Enum.map(fn(_) ->
  Task.async(fn ->
    Pooly.transaction("ChuckNorris", fn(worker_pid) ->
      ChuckFetcher.fetch(worker_pid)
    end, 5_000)
  end)
end)

tasks |> Enum.map(&Task.await(&1, 5_000))
```

TRANSACTIONS

```elixir
tasks = 1..5 |> Enum.map(fn(_) ->
  Task.async(fn ->
    Pooly.transaction("ChuckNorris", fn(worker_pid) ->
      ChuckFetcher.fetch(worker_pid)
    end, 5_000)
  end)
end)

tasks |> Enum.map(&Task.await(&1, 5_000))
```

TRANSACTIONS

```elixir
tasks = 1..5 |> Enum.map(fn(_) ->
  Task.async(fn ->
    Pooly.transaction("ChuckNorris", fn(worker_pid) ->
      ChuckFetcher.fetch(worker_pid)
    end, 5_000)
  end)
end)

tasks |> Enum.map(&Task.await(&1, 5_000))
```

TRANSACTIONS

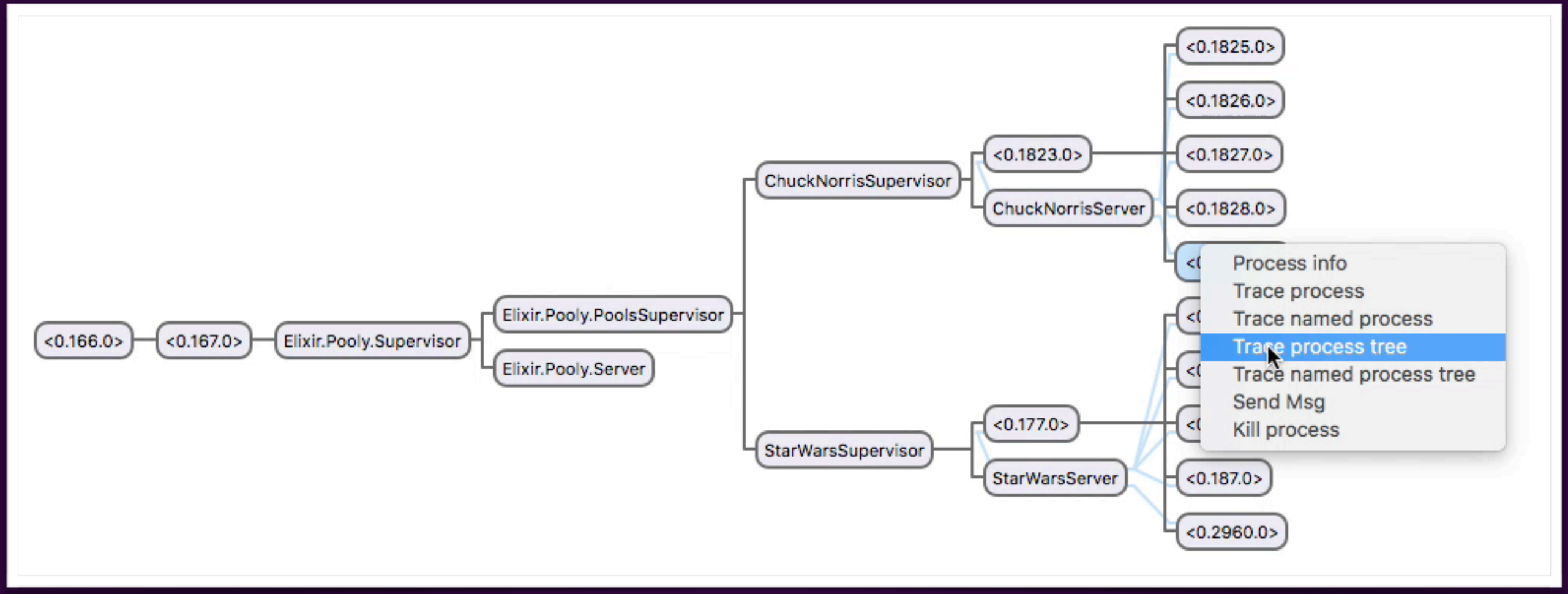
```elixir
tasks = 1..5 |> Enum.map(fn(_) ->
  Task.async(fn ->
    Pooly.transaction("ChuckNorris", fn(worker_pid) ->
      ChuckFetcher.fetch(worker_pid)
    end, 5_000)
  end)
end)

tasks |> Enum.map(&Task.await(&1, 5_000))
```

TRANSACTIONS

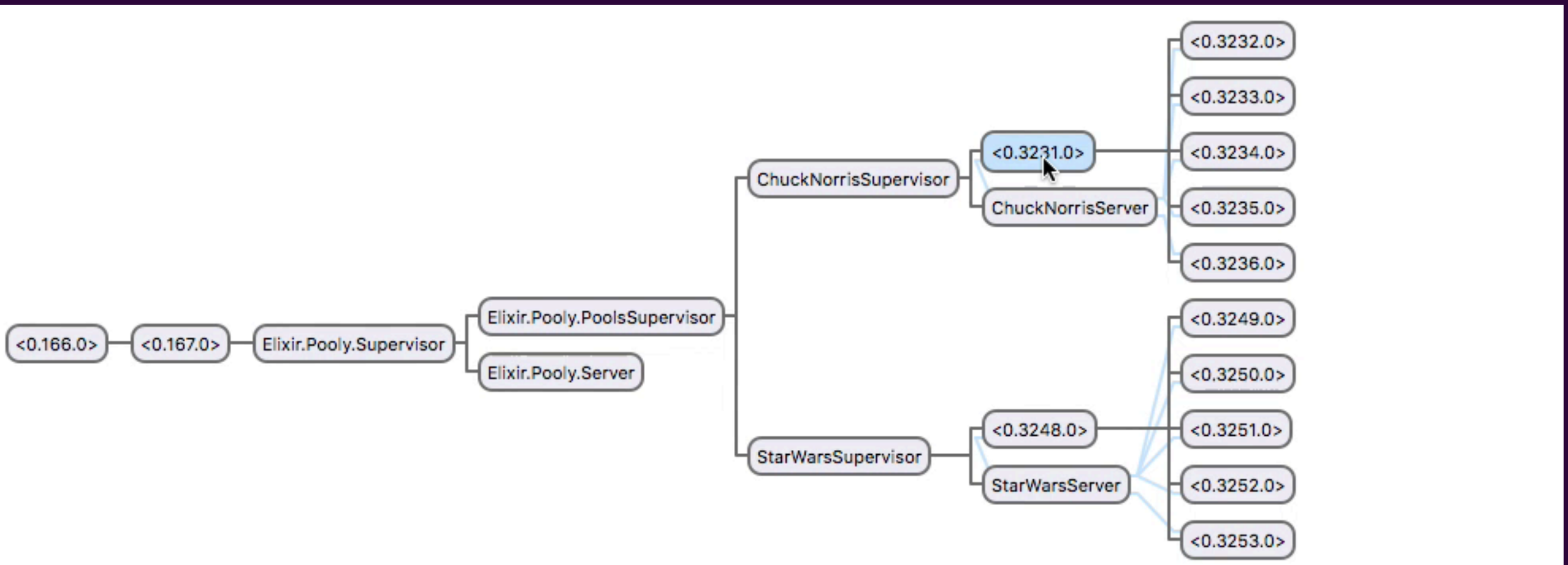
```elixir
tasks = 1..5 |> Enum.map(fn(_) ->
  Task.async(fn ->
    Pooly.transaction("ChuckNorris", fn(worker_pid) ->
      ChuckFetcher.fetch(worker_pid)
    end, 5_000)
  end)
end)

tasks |> Enum.map(&Task.await(&1, 5_000))
```

DEMO TIME!

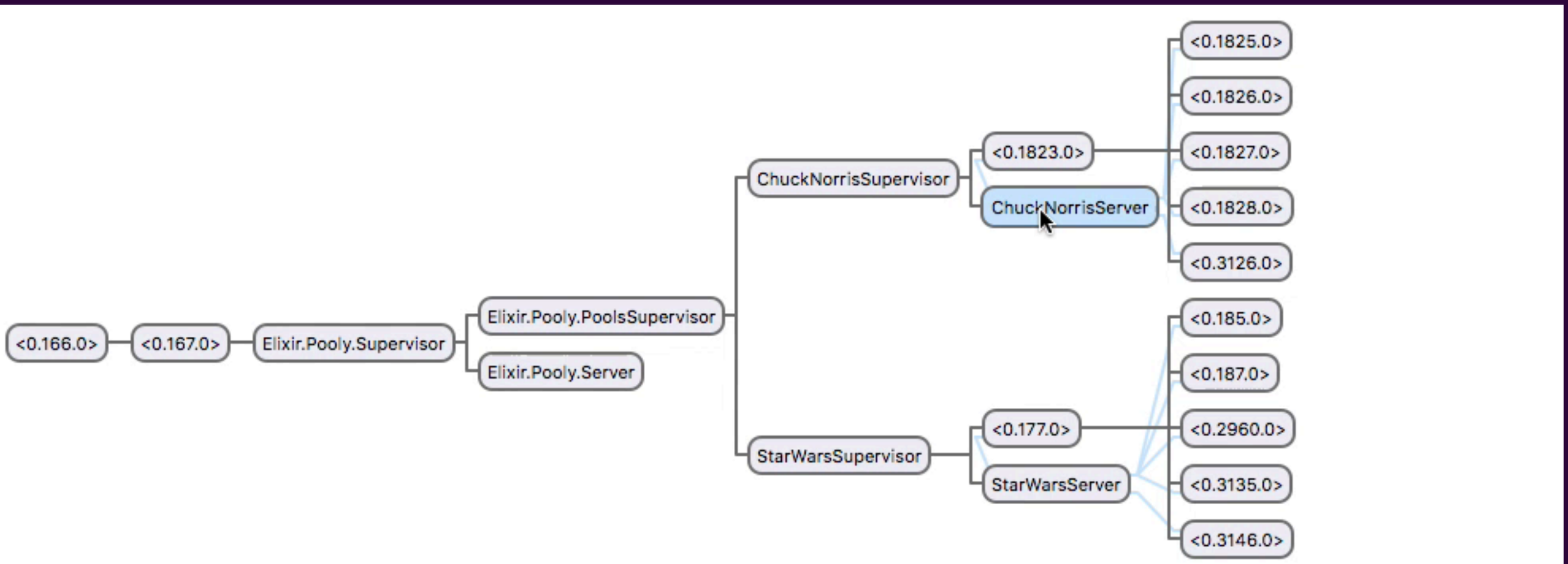
Demo: Killing Workers



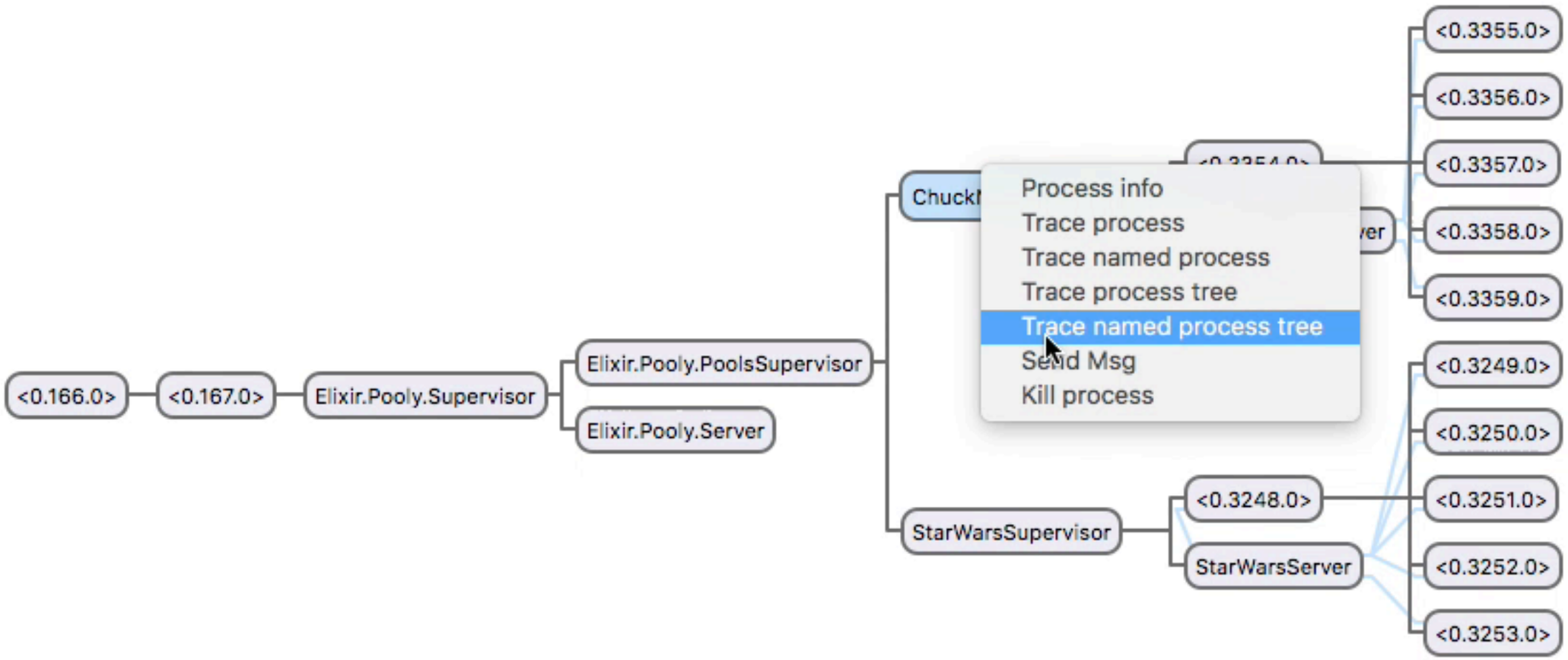
Demo: Killing Workers Supervisor



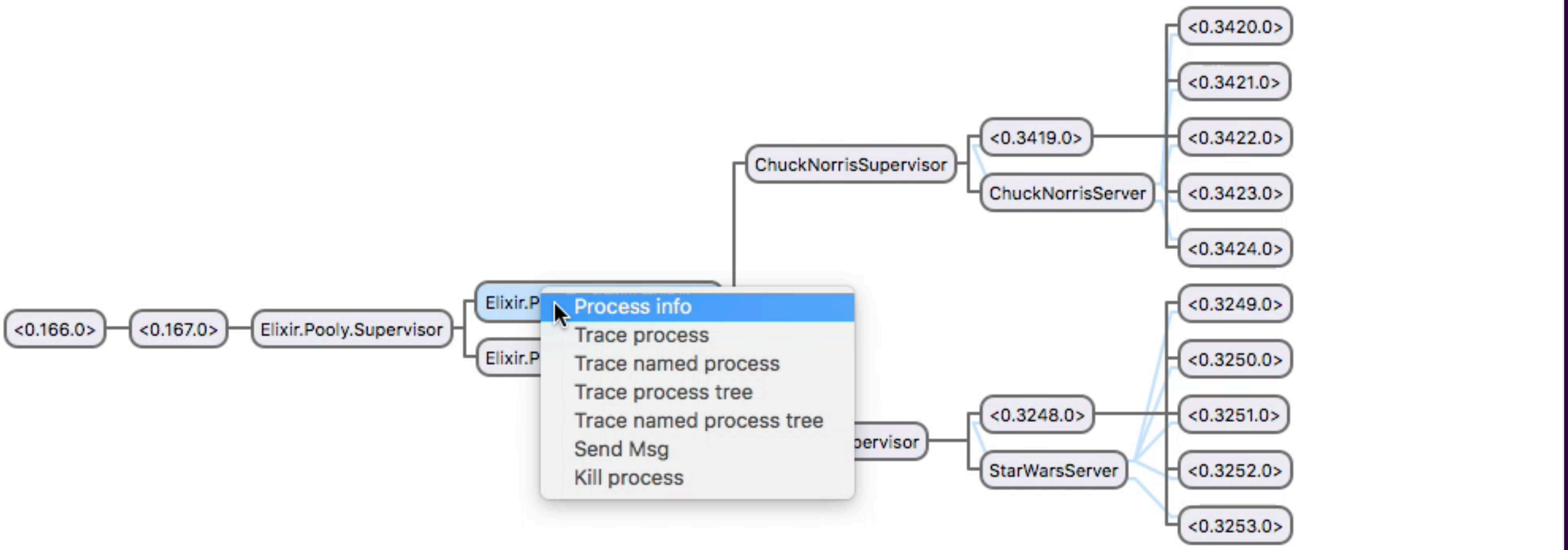
Demo: Killing Workers Supervisor



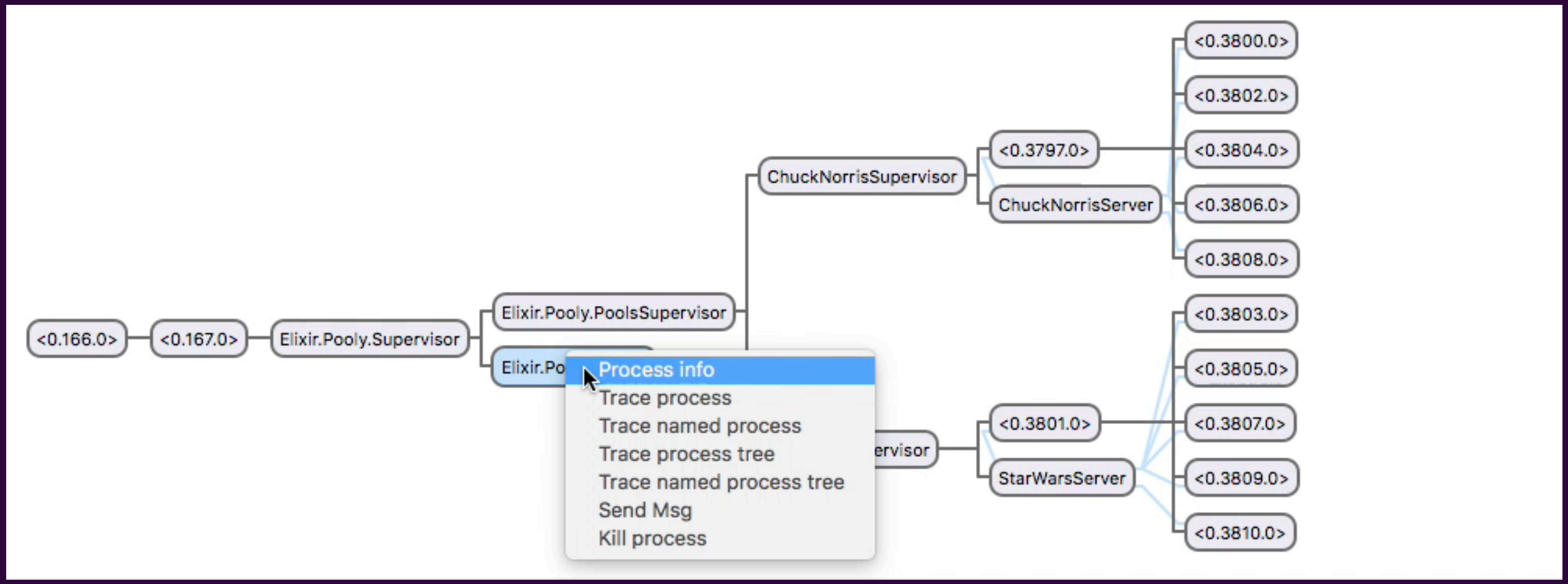
Demo: Killing Pool Supervisor



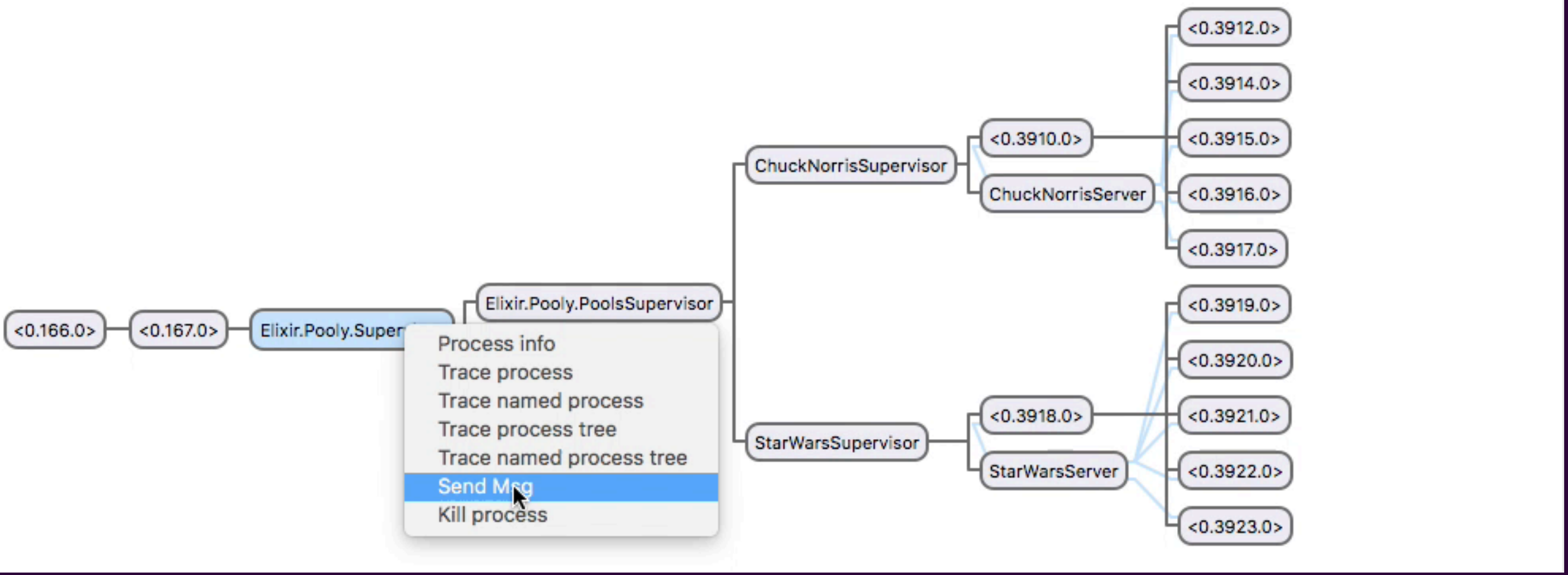
Demo: Killing Pools Supervisor



Demo: Killing Pools Server



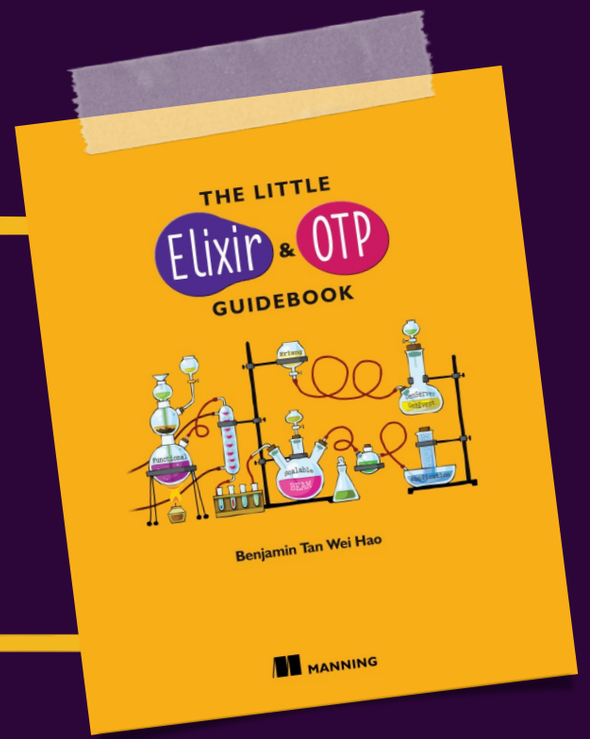
Demo: Killing The Top Level Supervisor




GREAT SUCCESS!

Resources

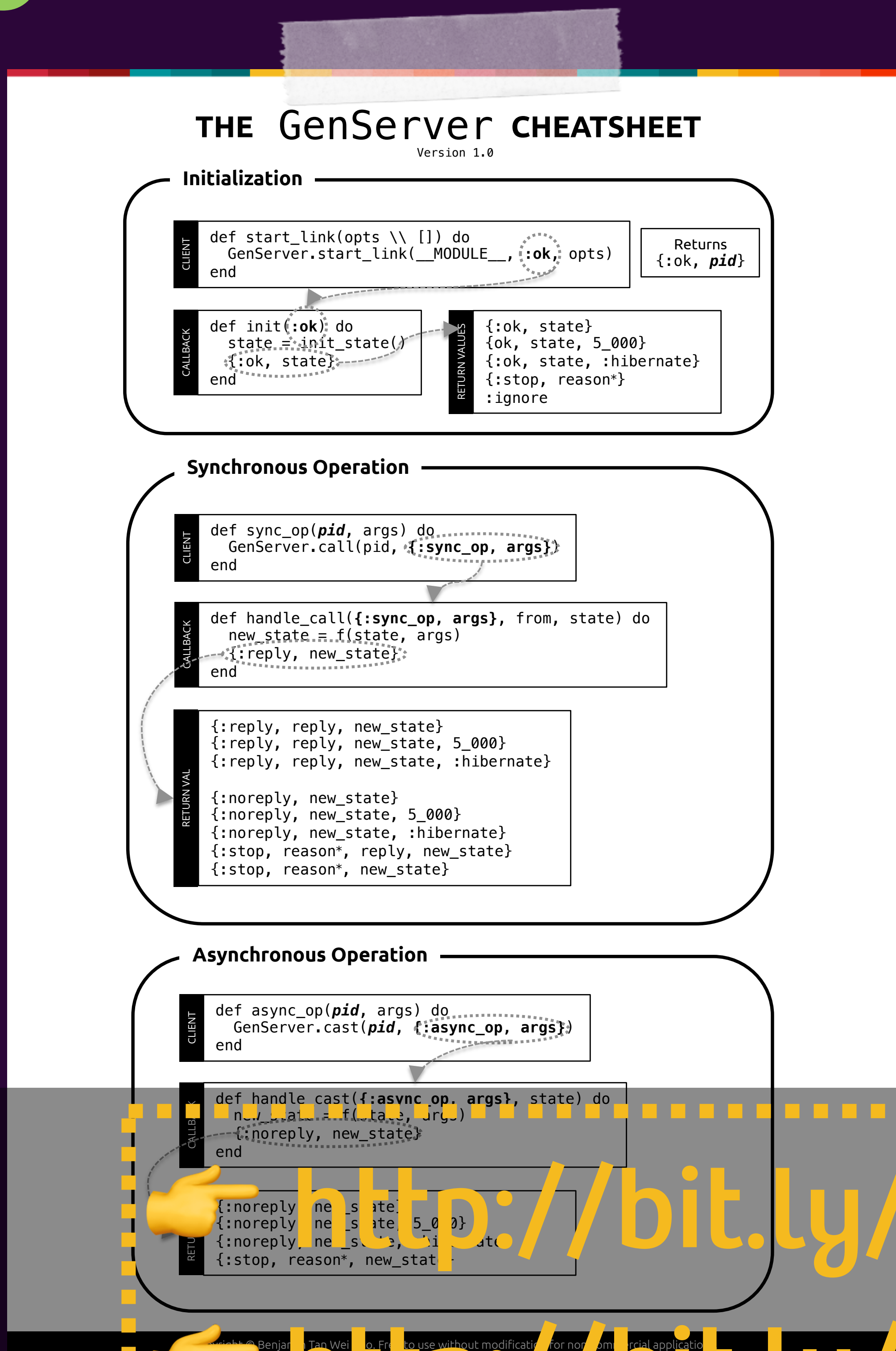
https://github.com/benjamintanweihao/the-little-elixir-otp-guidebook-code

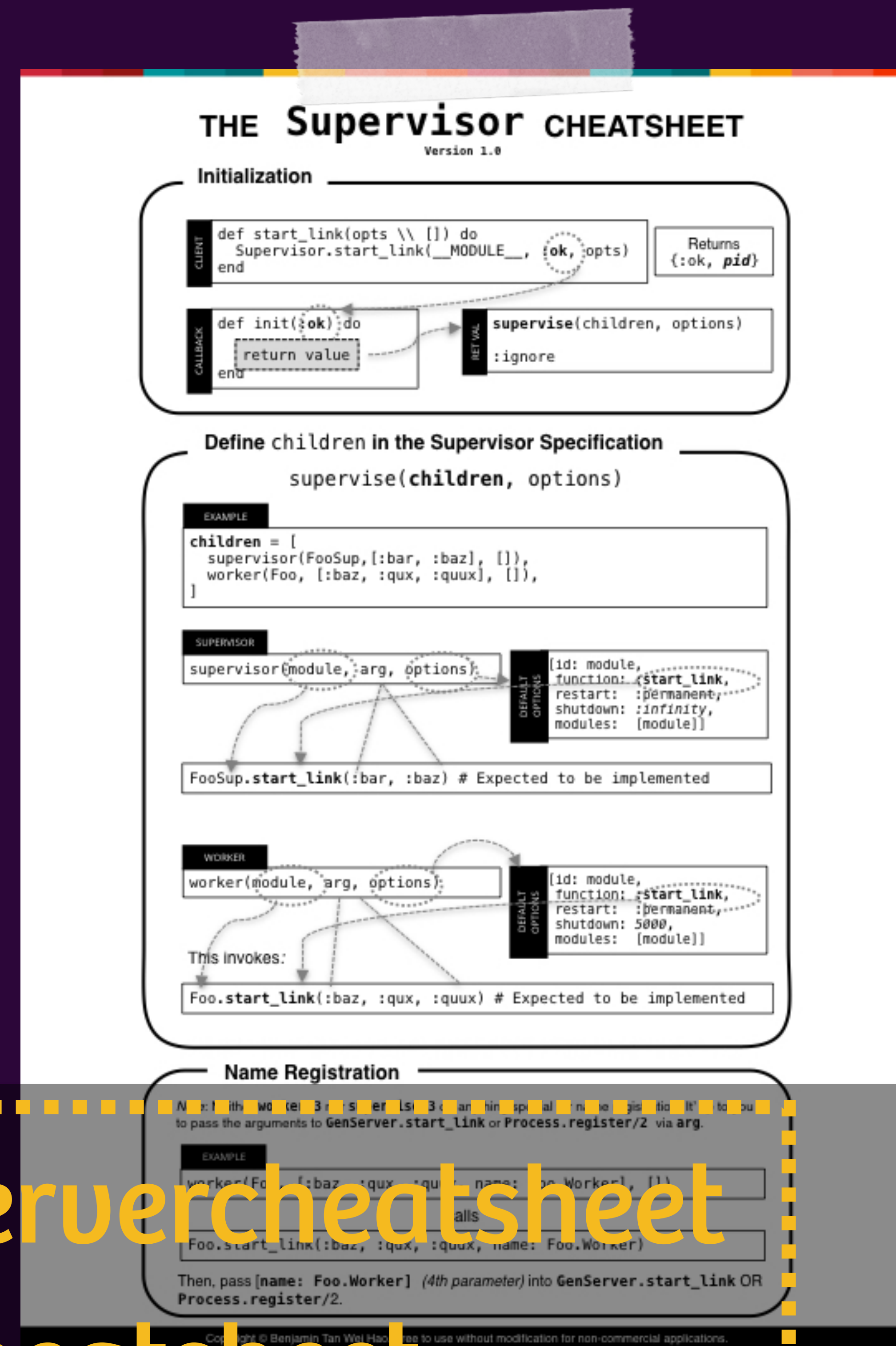


THE REAL THING™

https://github.com/devinus/poolboy

Resources



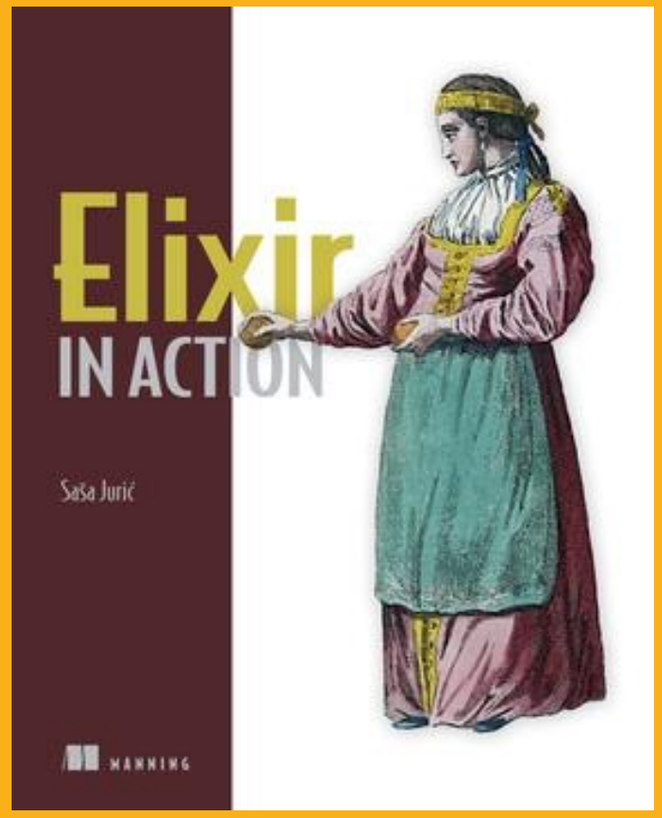


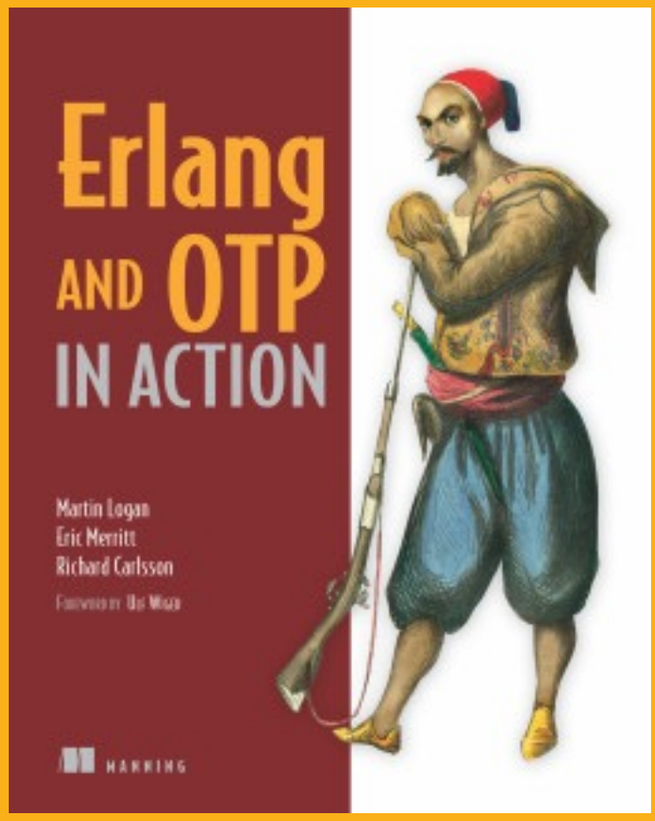
👉 http://bit.ly/genservercheatsheet

👉 http://bit.ly/supcheatsheet

ctweuc17

40% OFF!





KEEP

CALM

AND

LET IT

CRASH