Erlang/Elixir - Systeme, die laufen und laufen und laufen

Michael Sperber

Created: 2023-06-14 Wed 15:25

Erlang/Elixir - Systeme, die laufen und laufen und laufen

Michael Sperber

Active Group GmbH

@sperbsen@discuss.systems

Erlang

- Ericsson 1986
- erste Version implementiert in Prolog
- in Produktion seit 1995
- 1998 AXD301, Availability 99.9999999%, "nine nines"
- Open Source 1998
- virtuelle Maschine BEAM

Wofür Erlang?

- nebenläufig
- verteilt
- fehlertolerant
- "soft real-time"
- hochverfügbar
- unterbrechungsfrei
- "Hot Swap"
- embedded

Funktionen und Atome

```erlang
-spec double(number()) -> number().
double(X) -> X * 2.

% Ist ein Haustier niedlich?
-spec is_cute(dog | cat | snake) -> boolean().
is_cute(dog) -> true;
is_cute(cat) -> true;
is_cute(snake) -> false.
```

Tupel

```erlang
-spec safe_divide(number(), number())
                  -> {error, divide_by_zero} | {ok, number()}.
safe_divide(X, Y) ->
    if
        Y == 0 -> {error, divide_by_zero};
        true -> {ok, X / Y}
    end.
```

```erlang
% Steigung eine Geraden berechnen
-spec slope({number(), number()}, {number(), number()})
            -> number() | vertical.
slope({X1, Y1}, {X2, Y2}) ->
    case safe_divide(Y2 - Y1, X2 - X1) of
        {error, divide_by_zero} -> vertical;
        {ok, S} -> S
    end.
```

Records

```
% Ein Gürteltier hat folgende Eigenschaften:
% - tot oder lebendig - UND -
% - Gewicht
-record(dillo,
        { liveness :: dead | alive,
          weight :: number()}).

d1() -> #dillo{liveness = alive, weight = 10}.
d2() -> #dillo{liveness = dead, weight = 8}.

% Gürteltier überfahren
-spec run_over_dillo(#dillo{}) -> #dillo{}.
run_over_dillo(#dillo{ weight = W }) ->
   #dillo{ liveness = dead, weight = W}.
```

Elixir

- Beginn 2012
- José Valim
- beinflusst von Ruby und Rails

Funktionen und Atome

```elixir
@spec double(number()) :: number()
def double(x) do x * 2 end

@spec isCute(Dog | Cat | Snake) :: boolean()
def isCute(Dog) do true end
def isCute(Cat) do true end
def isCute(Snake) do false end
```

Tupel

```elixir
@spec safeDivide(number(), number())
        :: {:error, :divide_by_zero} | {:ok, number()}
def safeDivide(x, y) do
  if y == 0 do
     {:error, :divide_by_zero}
  else
     {:ok, x / y}
  end
end

@spec slope({number(), number()}, {number(), number()})
        :: number() | :vertical
def slope({x1, y1}, {x2,y2}) do
  case safeDivide(y2-y1, x2-x1) do
    {:error, :divide_by_zero} -> :vertical
    {:ok, s} -> s
  end
end
```

Structs

```elixir
defmodule Dillo do
  @enforce_keys [:liveness, :weight]
  defstruct [:liveness, :weight]
  @type t :: %Dillo{liveness: Dead | Alive, weight: number()}
  @spec make(Dead | Alive, number()) :: Dillo.t
  def make(liveness, weight) do
    %Dillo{liveness: liveness, weight: weight}
  end

  @doc """
  Gürteltier überfahren

  iex> Animal.Dillo.run_over(Animal.Dillo.make(Alive), 10))
  %Animal.Dillo{alive?: Dead, weight: 10}
  """
  @spec run_over(Dillo.t) :: Dillo.t
  def run_over(dillo) do
    Dillo.make(Dead, dillo.weight)
  end
end
```

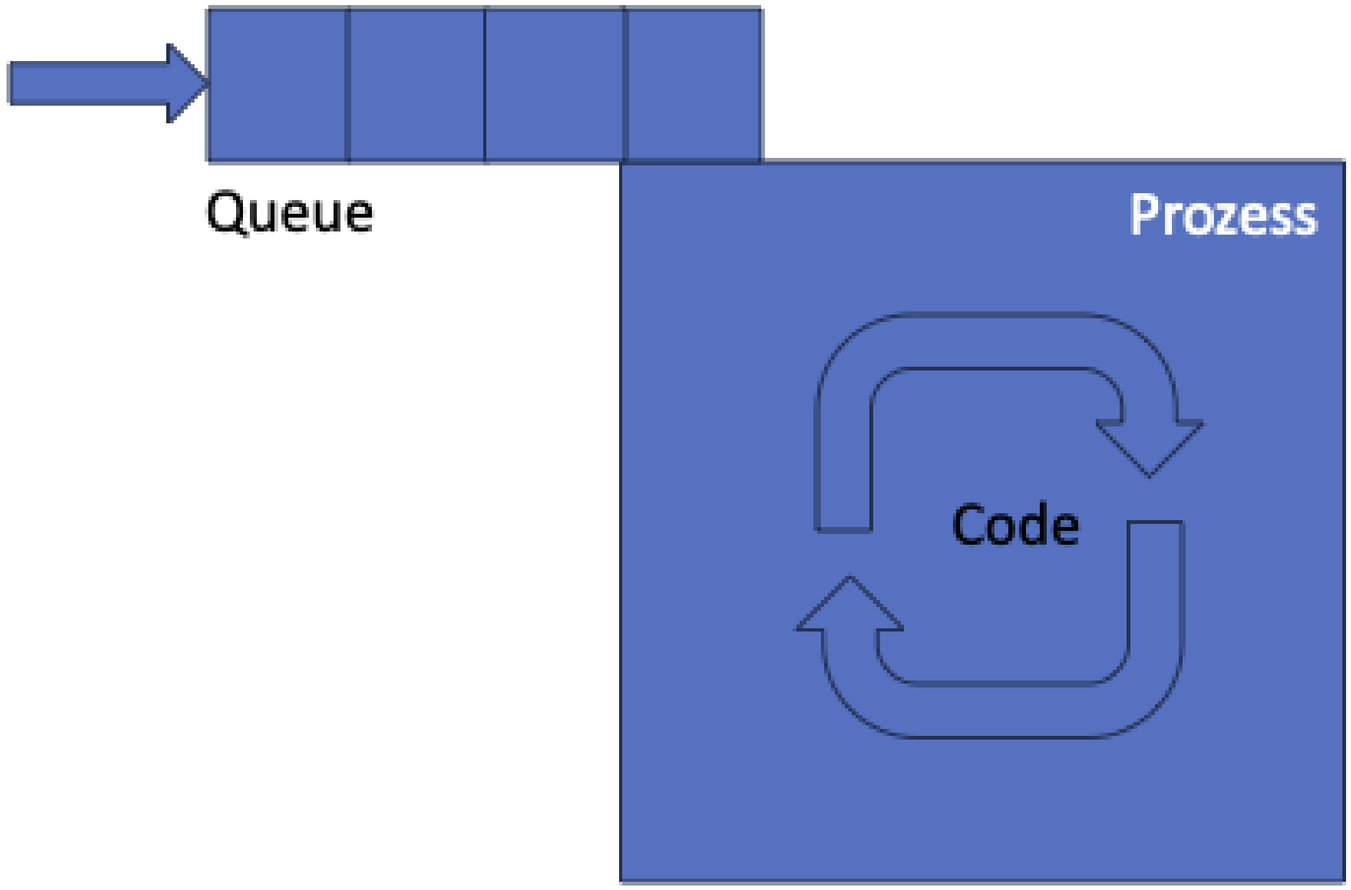
Quick Structs

```elixir
defmodule Dillo do
  use QuickStruct, [liveness: Dead | Alive, weight: number()]

  @doc """
  Gürteltier überfahren

  iex> Animal.Dillo.run_over(Animal.Dillo.make(Alive), 10))
  %Animal.Dillo{alive?: Dead, weight: 10}
  """
  @spec run_over(Dillo.t):: Dillo.t
  def run_over(dillo) do
    Dillo.make(Dead, dillo.weight)
  end
end
```

Unterschiede Erlang / Elixir

- rebar3 vs. mix
- strings vs. Strings
- Phoenix

Prozesse auf der BEAM



Prozesse

```
def processCode(n) do
  receive do
    Mike ->
      IO.puts("super #{n}")
      processCode(n)
    Sperber ->
      IO.puts("doof")
  end
```

Zähler-Prozess

```
defmodule IncBy do
  use QuickStruct, [increment: integer()]
end
defmodule Get do
  use QuickStruct, [requester: pid() ]
end

def counterCode(n) do
  IO.puts("counter: #{n}")
  receive do
    Inc -> counterCode(n+1)
    %IncBy{increment: inc} -> counterCode(n+inc)
    %Get{requester: req} ->
      send(req, n)
      counterCode(n)
  end
end
```

Zähler-Prozess

```
def startCounter(n) do
  spawn(fn () -> counterCode(n) end)
end

def counterInc(pid) do
  send(pid, Inc)
  :ok
end
```

```
def counterInc(pid, inc) do
  send(pid, IncBy.make(inc))
  :ok
end

def counterGet(pid) do
  send(pid, Get.make(self()))
  receive do
    n -> n
  end
end
```

Open Telephony Platform (OTP)

```elixir
defmodule Counter do
  use GenServer

  defmodule IncBy do
    use QuickStruct, [increment: integer()]
  end
  defmodule Get do
    use QuickStruct, [requester: pid() ]
  end

  @type message() :: Inc | IncBy.t() | Get.t()

  @type state() :: integer()
  ...
end
```

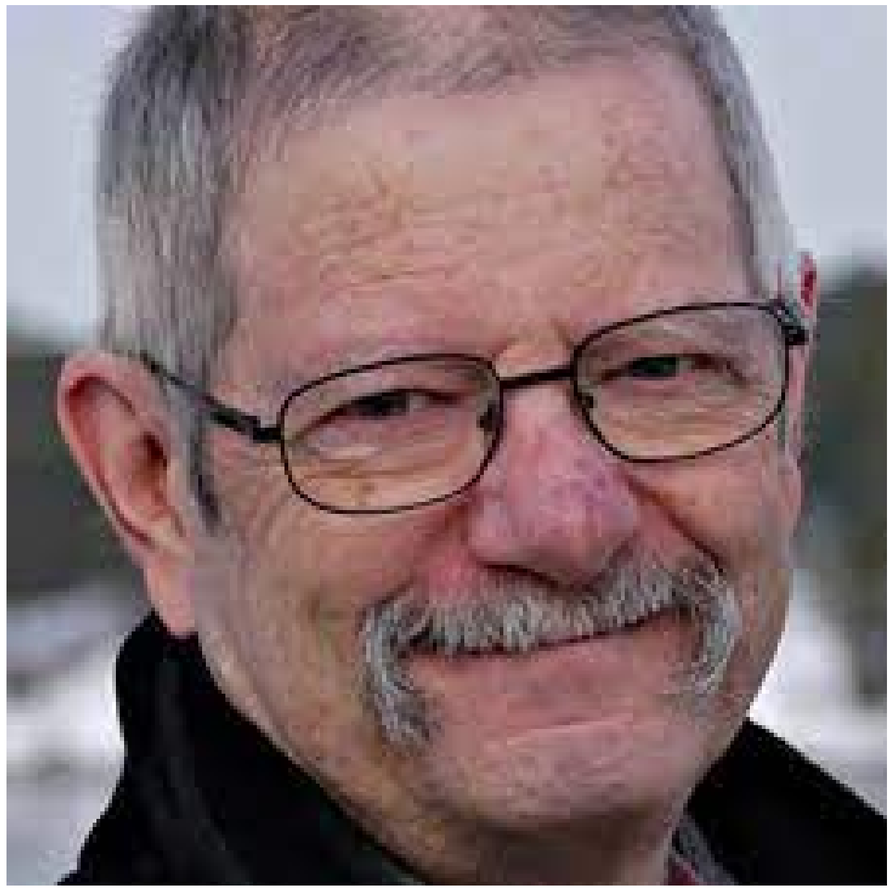
Prozesse

```
defmodule Counter
  ...
  @spec init(state()) :: {:ok, state()}
  def init(initialState) do
    {:ok, initialState}
  end

  @spec handle_cast(message(), state()) :: {:noreply, state()}
  def handle_cast(Inc, n) do
    {:noreply, n+1}
  end

  @spec handle_call(message(), GenServer.from(), state())
    :: {:reply, integer(), state()}
  def handle_call(%Get{requester: _req}, _from, n) do
    {:reply,
     n, # reply
     n} # new state
  end
  ...
end
```

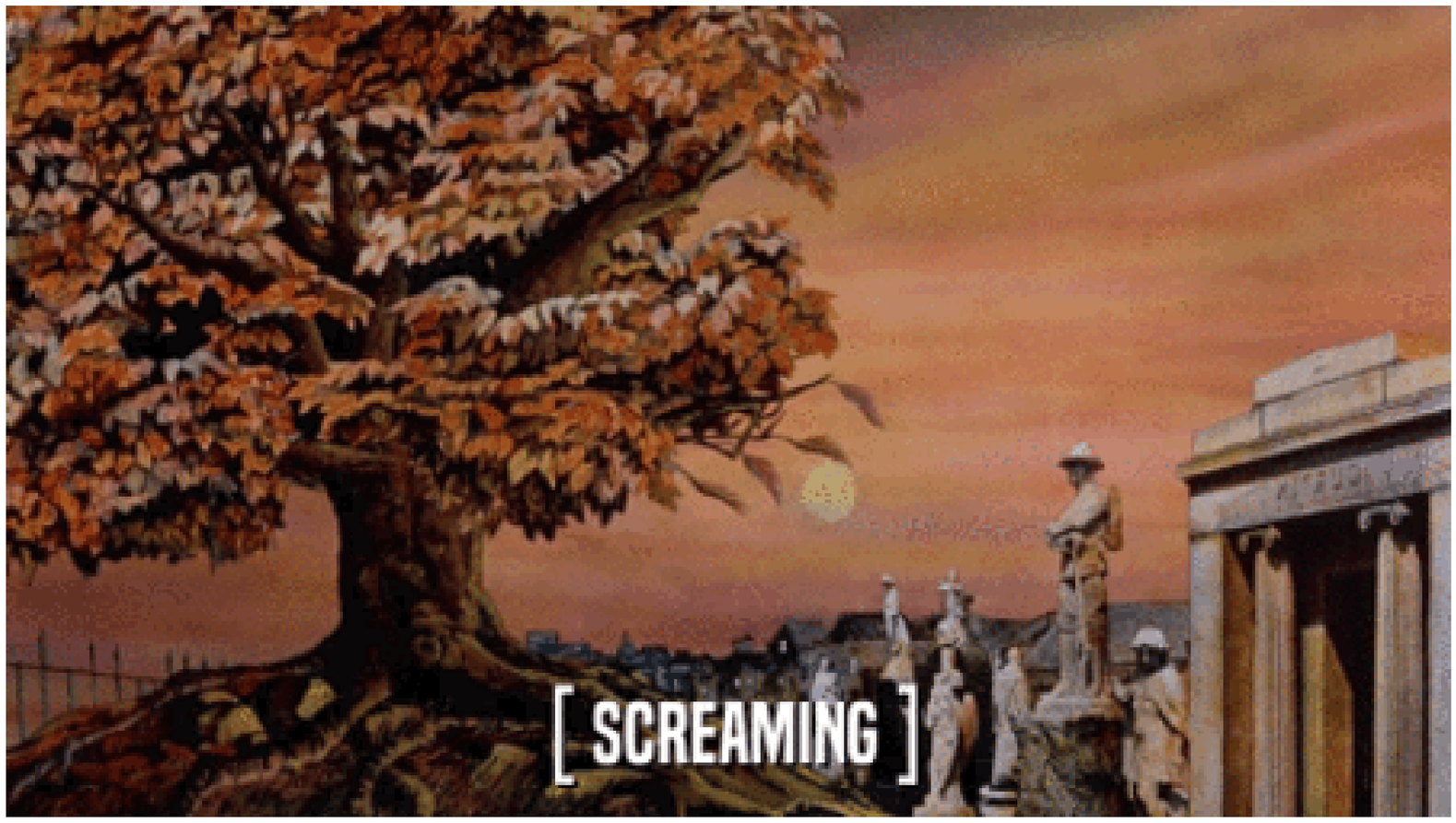
Prozesse

```
defmodule Counter
  ...
  def start(initialN) do
    GenServer.start(Counter, initialN, [{:debug, [:trace]}])
  end

  def counterInc(pid, inc) do
    GenServer.cast(pid, IncBy.make(inc))
  end

  def counterGet(pid) do
    GenServer.call(pid, Get.make(self()))
  end
end
```

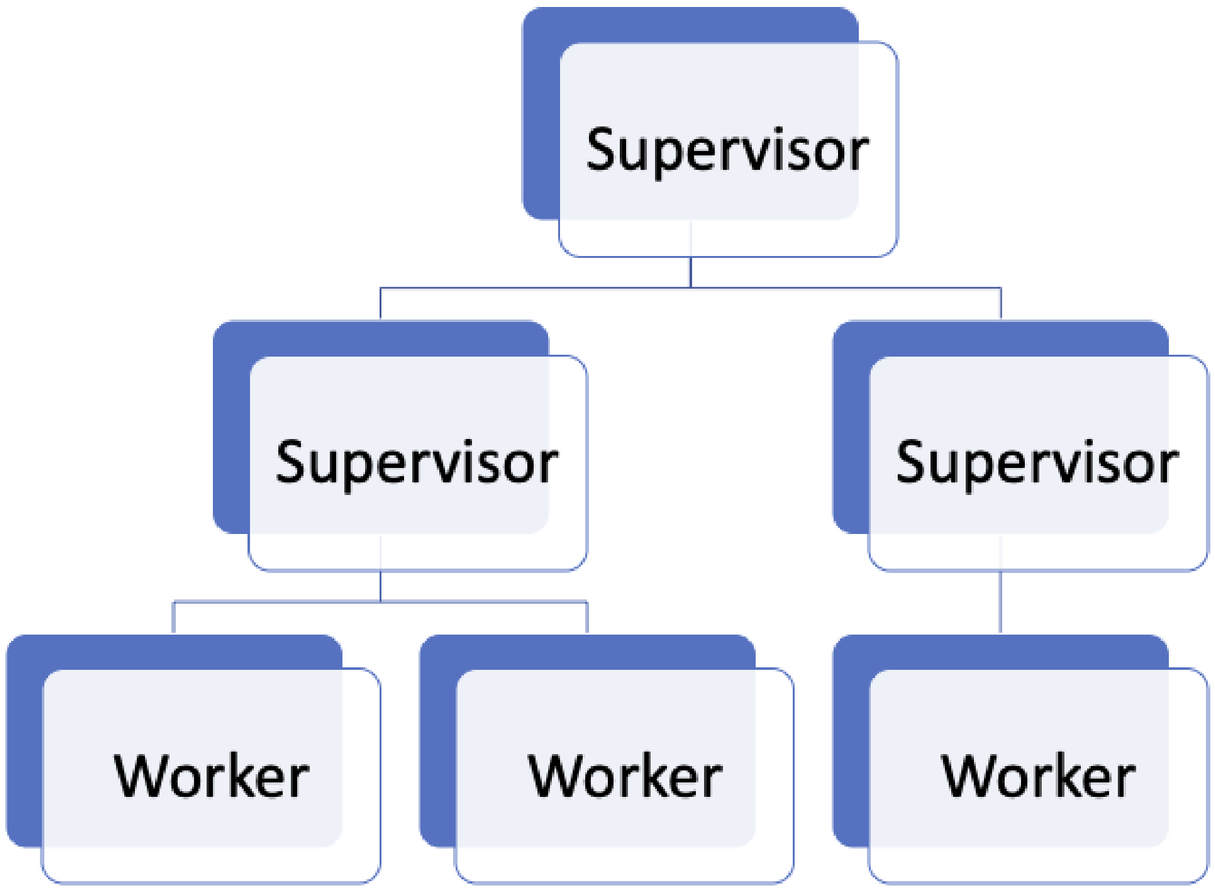
Let It Crash



Prozesse verlinken



Supervisoren



Bankkonten

```
defmodule Bank.Accounts do
  @type t :: %{String.t => integer()}

  def empty() do
      %{}
  end
  ...
end
```

Bankkonto erzeugen

```elixir
@spec create(Bank.Accounts.t, String.t)
  :: {:ok, t} | :account_exists
def create(accounts, name) do
  if Map.get(accounts, name) do
    :account_exists
  else
    {:ok, Map.put(accounts, name, 0)}
  end
end
```

Bankkonto - Einzahlen

```elixir
@spec deposit(Bank.Accounts.t, String.t, pos_integer())
  :: {:ok, t} | :no_account
def deposit(accounts, name, amount) do
  previous_amount = Map.get(accounts, name)
  if previous_amount do
    {:ok, Map.put(accounts, name, previous_amount + amount)}
  else
    :no_account
  end
end
```

Bankkonto - Abheben

```elixir
@spec withdraw(Bank.Accounts.t, String.t, pos_integer())
  :: {:ok, t} | :no_account
def withdraw(accounts, name, amount) do
  previous_amount = Map.get(accounts, name)
  if previous_amount do
    {:ok, Map.put(accounts, name, previous_amount - amount)}
  else
    :no_account
  end
end
```

Bankkonto - Abfragen

```elixir
@spec balance(Bank.Accounts.t, String.t)
  :: {:ok, integer()} | :not_found
def balance(accounts, name) do
  balance = Map.get(accounts, name)
  if balance do
    {:ok, balance}
  else
    :not_found
  end
end
```

Einfache Bank - OTP

```
defmodule Bank.Simple do
  use GenServer
  alias Bank.Command.{Balance, Withdraw, Deposit, Create}
  alias Bank.Accounts

  def balance(name) do
    GenServer.call(:simple_bank, Balance.make(name))
  end
  def withdraw(name, amount) do
    GenServer.cast(:simple_bank, Withdraw.make(name, amount))
  end
  def deposit(name, amount) do
     GenServer.cast(:simple_bank, Deposit.make(name, amount))
  end
  def create(name) do
    GenServer.cast(:simple_bank, Create.make(name))
  end
  ...
end
```

Einfache Bank - OTP

```elixir
def start() do
  GenServer.start(__MODULE__, [], name: :simple_bank)
end

def stop() do
  GenServer.stop(:simple_bank)
end
```

```elixir
def init(_) do
  {:ok, Accounts.empty()}
end
```

Einfache Bank - Handler

```
@spec handle_cast(Bank.Command.write, Accounts.t)
  :: {:noreply, Accounts.t}

def handle_cast(%Create{name: name}, state) do
  case Accounts.create(state, name) do
      {:ok, new_state} -> {:noreply, new_state}
      :account_exists -> {:noreply, state}
  end
end
```

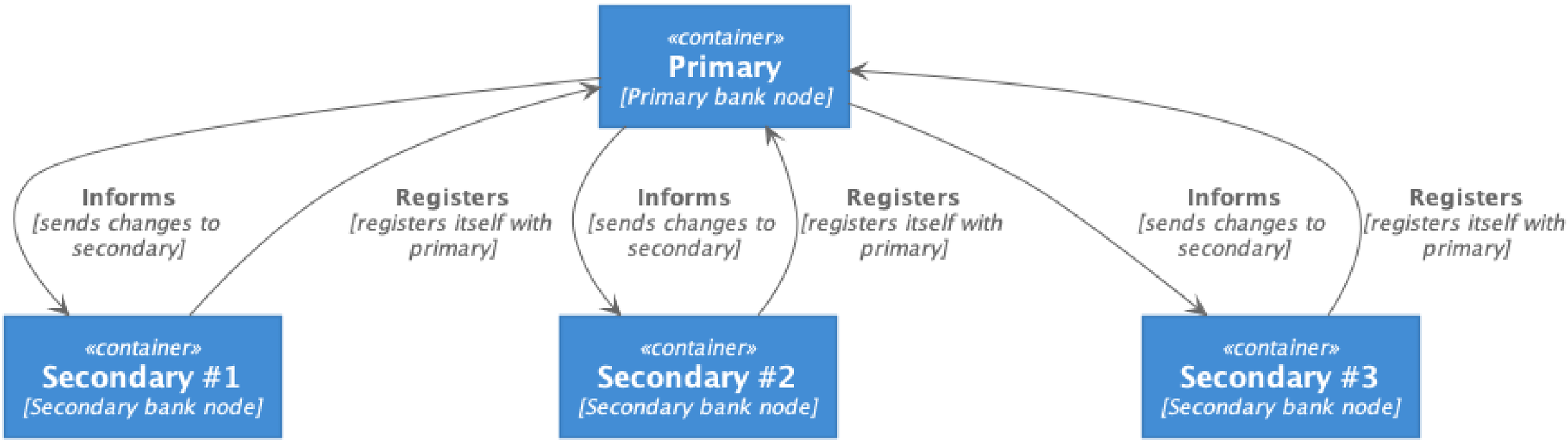
Einfache Bank - Handler

```
def handle_cast(%Deposit{name: name, amount: amount}, state) do
  case Accounts.deposit(state, name, amount) do
       {:ok, new_state} -> {:noreply, new_state}
       :no_account -> {:noreply, state}
  end
end

def handle_cast(%Withdraw{name: name, amount: amount}, state) do
  case Accounts.withdraw(state, name, amount) do
       {:ok, new_state} -> {:noreply, new_state}
       :no_account -> {:noreply, state}
  end
end
```

Einfache Bank - Abfrage

```
@spec handle_call(Bank.Command.read, any(), Accounts.t)
  :: {:reply, {:ok, integer()} | :not_found, Accounts.t}

def handle_call(%Balance{name: name}, _ , state) do
  case Accounts.balance(state, name) do
      {:ok, balance} -> {:reply, {:ok, balance}, state}
      :not_found -> {:reply, :not_found, state}
  end
```

Verteilung


«container»
Primary
[Primary bank node]
Informs
[sends changes to secondary]
Registers
[registers itself with primary]
Informs
[sends changes to secondary]
Registers
[registers itself with primary]
Informs
[sends changes to secondary]
Registers
[registers itself with primary]
«container»
Secondary #1
[Secondary bank node]
«container»
Secondary #2
[Secondary bank node]
«container»
Secondary #3
[Secondary bank node]

Verteilung - Primary

```elixir
defmodule Bank.Primary do

  use GenServer
  require Logger

  alias Bank.Command.{Balance, Create, Withdraw, Deposit, Register}
  alias Bank.Accounts

  defmodule PrimaryState do
    use QuickStruct, [accounts: Accounts.t, secondaries: [pid()]]
  end

  def init(_), do: {:ok, PrimaryState.make(Accounts.empty(), [])}

  ...
end
```

Verteilung - Primary - Stubs

```elixir
def primary_name(), do: {:global, :Primary}

def balance(name) do
   GenServer.call(primary_name(), Balance.make(name))
end
def withdraw(name, amount) do
   GenServer.cast(primary_name(), Withdraw.make(name, amount))
end
def deposit(name, amount) do
  GenServer.cast(primary_name(), Deposit.make(name, amount))
end
def create(name) do
  GenServer.cast(primary_name(), Create.make(name))
end
```

Verteilung - Multicast

```elixir
def broadcast(secondaries, message) do
  secondaries
  |> Enum.each(fn secondary ->
                 GenServer.cast(secondary, message)
               end)
end
```

Verteilung - Primary - Kontenerzeugung

```
def handle_cast(%Create{name: name} = command, state) do
  broadcast(state.secondaries, command)
  case Accounts.create(state.accounts, name) do
      {:ok, new_accounts} ->
        {:noreply, %{state | accounts: new_accounts}}
      :account_exists -> {:noreply, state}
  end
end
```

Verteilung - Primary - Transaktionen

```
def handle_cast(%Deposit{name: name, amount: amount} = command,
                state) do
  broadcast(state.secondaries, command)
  case Accounts.deposit(state.accounts, name, amount) do
      {:ok, new_accounts} ->
        {:noreply, %{state | accounts: new_accounts}}
      :no_account -> {:noreply, state}
  end

def handle_cast(%Withdraw{name: name, amount: amount} = command,
                state) do
  broadcast(state.secondaries, command)
  case Accounts.withdraw(state.accounts, name, amount) do
      {:ok, new_accounts} ->
        :noreply, %{state | accounts: new_accounts}}
      :no_account -> {:noreply, state}
  end
end
```

Verteilung - Primary - Kontostand

```
def handle_call(%Balance{name: name}, _ ,  state) do
  case Accounts.balance(state.accounts, name) do
       {:ok, balance} -> {:reply, {:ok, balance}, state}
       :not_found -> {:reply, :not_found, state}
  end
end
```

Verteilung - Primary - Secondaries

```elixir
def handle_call(%Register{pid: pid}, _, state) do
  Process.monitor(pid)
  {:reply, {:ok, state.accounts},
    %{state | secondaries: [pid | state.secondaries]}}
end


def handle_info({:DOWN, ref, _, pid, reason},
                %PrimaryState{secondaries: secondaries} = state) do
  Process.demonitor(ref)
  {:noreply, %{state | secondaries: List.delete(secondaries, pid)}}
end
```

Verteilung - Secondary

```elixir
defmodule Bank.Secondary do
  ...
  def balance(bank, name), do
    GenServer.call(bank, Balance.make(name))
  end
  def withdraw(name, amount) do
    Primary.withdraw(name, amount)
  end
  def deposit(name, amount) do
    Primary.deposit(name, amount)
  end
  def create(name), do: Primary.create(name)
  ...
end
```

Verteilung - Secondary - Delegation

```
def handle_cast(%Create{name: name}, state) do
  case Accounts.create(state, name) do
       {:ok, new_state} -> {:noreply, new_state}
       :account_exists -> {:noreply, state}
  end
end
```

Verteilung - Secondary

```
def handle_call(%Balance{name: name}, _ , state) do
  case Accounts.balance(state, name) do
       {:ok, balance} -> {:reply, {:ok, balance}, state}
       :not_found -> {:reply, :not_found, state}
  end
end
```

Verteilung - Supervisor

```
defmodule Bank.Super do
  def children() do
    [
      %{
        id: :primary,
        start: {Bank.Primary, :start_link, []}
      },
      %{
        id: :secondaries,
        start: {Bank.Super, :start_children, []}
      }
    ]
  end
  ...
end
```

Verteilung - Supervisor

```elixir
def secondary_children() do
  [
    %{
      id: :secondary1s,
      start: {Bank.Secondary, :start_link, [:secondary1]}
    },
    %{
      id: :secondary2s,
      start: {Bank.Secondary, :start_link, [:secondary2]}
    }
  ]
end
```

Verteilung - Supervisor

```
def start_link() do
  Supervisor.start_link(children(), strategy: :rest_for_one)
end

def start_children() do
  Supervisor.start_link(secondary_children(), strategy: :one_for_one
end
```

Hot Update

```
defmodule HotUpdate do
  use QuickStruct, [replacement: (() -> any())]
end

def processCode(n) do
  receive do
    Mike ->
      IO.puts("super #{n}")
      processCode(n)
    Sperber ->
      IO.puts("doof")
    %HotUpdate{replacement: f} -> f.()
  end
end
```

Was gibt's noch

- Dialyzer
- Elixir: Phoenix
- Deployment
- Rollout
- Monitoring
- Observability
- alles, was man sonst braucht

Aufpassen

- make Erlang/Elixir, not CRUD
- make Elixir, not Rails
- verteilt ≠ lokal

Zusammenfassung

- Elixir super
- Erlang super
- läuft
- läuft
- läuft

Funktionale Softwarearchitektur

- Tübingen
- Individualsoftware
- Scala, Clojure, F#, Haskell, OCaml, Erlang, Elixir, Swift
- Schulungen (iSABQ Foundation, FUNAR, FLEX mit Erlang Elixir)
- auch hier: Kaan Sahin (Clojure), Markus Schlegel (UI-Design und morgen Kopplung und Sabotage)

Blog: https://funktionale-programmierung.de