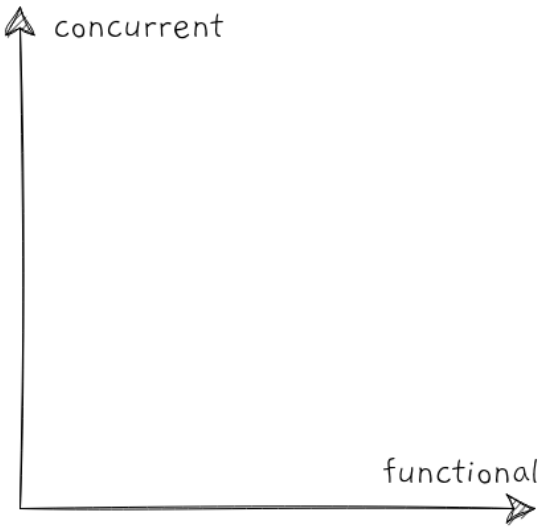
Thinking in Elixir

@sasajuric


concurrent
functional

thinking in

functional Elixir
concurrent Elixir

functional Elixir

functions & modules
immutable data
first-class functions

```
some_var = "Hello, World!"
some_var.length
```

```
String.length(some_var)
```

```
a_map = {"foo" => 1, "bar" => 2}
a_map.delete("bar")
```

```
a_map = %{"foo" => 1, "bar" => 2}
another_map = Map.delete(a_map, "bar")
```

```elixir
a_map = %{"foo" => 1, "bar" => 2}
a_map = Map.delete(a_map, "bar")
```

blackjack

```
deck = Deck.shuffled()
{card1, deck} = Deck.pop(deck)
{card2, deck} = Deck.pop(deck)
```

```
def shuffled do
  suits = [:spades, :diamonds, :clubs, :hearts]
  ranks = [2, 3, 4, 5, 6, 7, 8, 9, 10, :jack, :queen, :king, :ace]

  all_cards =
    for suit <- suits,
        rank <- ranks,
        do: %{rank: rank, suit: suit}

  Enum.shuffle(all_cards)
end
```

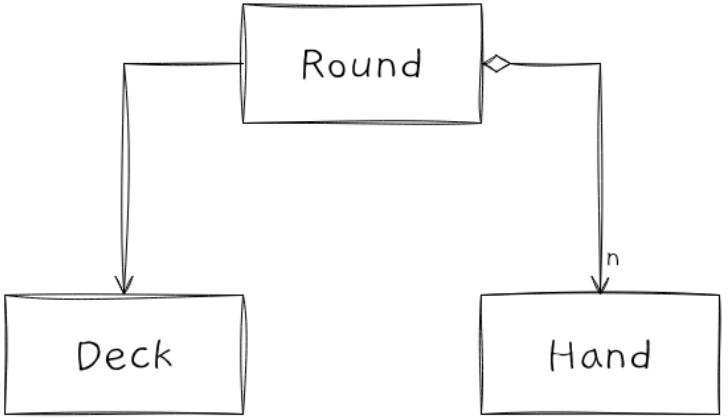
```
def pop([top_card | rest]), do: {top_card, rest}
def pop([]), do: pop(shuffled())
```

```
hand = Hand.new()

hand = Hand.add(hand, card1)
hand = Hand.add(hand, card2)

Hand.score(hand)

Hand.busted?(hand)
```

```
def new,
  do: %Hand{cards: [], valid_scores: [0]}
```

```
def add(hand, card) do
  cards = [card | hand.cards]

  valid_scores =
    for score <- hand.valid_scores,
        card_value <- card_values(card.rank),
        score = score + card_value,
        score <= 21,
        do: score

  %Hand{hand | cards: cards, valid_scores: valid_scores}
end
```

```
deck = Deck.shuffled()
hand = Hand.new()

{card, deck} = Deck.pop()
hand = Hand.add(hand, card)
```


Round
Deck
Hand
n

```
players = ["Péter", "Viktor", "Saša"]
deck = Deck.new()
round = Round.new(players, deck)
```

```
Round.current_player(round)
# Péter

round = Round.hit(round)
round = Round.stay(round)

Round.current_player(round)
# Viktor
```

```
Round.winners(round)
# ["Péter"]
```

```elixir
def new(players, deck) do
  hands = Enum.map(players, &Hand.new/1)
  {hands, deck} = deal_initial_cards(hands, deck)
  %Round{deck: deck, hands: hands, finished: []}
end
```

```
def stay(round) do
  [hand | other_hands] = round.hands
  finished = [hand | round.finished]
  %Round{round | hands: other_hands, finished: finished}
end
```

```
def hit(round) do
  {card, deck} = Deck.pop(round.deck)

  [hand | other_hands] = round.hands
  hand = Hand.add(hand, card)
  hands = [hand | other_hands]

  round = %Round{round | deck: deck, hands: hand}

  if Hand.busted?(hand), do: stay(round), else: round
end
```

```elixir
def winners(round) do
  round.finished
  |> Enum.reject(&Hand.busted?/1)
  |> Enum.group_by(&Hand.score/1)
  |> Enum.max(fn -> {nil, []} end)
  |> then(fn {_score, winning_hands} -> winning_hands end)
  |> Enum.map(& &1.player)
end
```

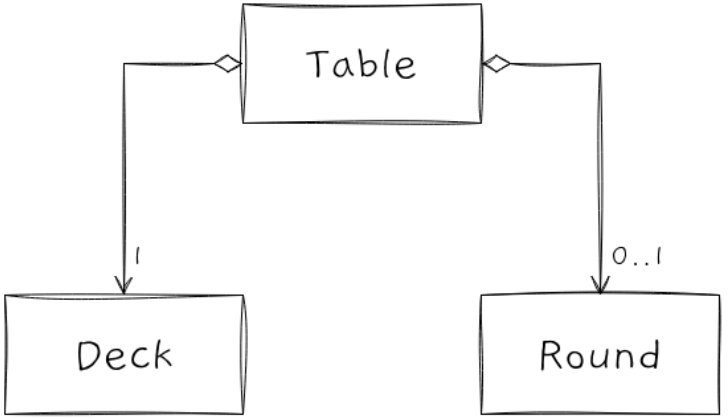


Table
1
0..1
Deck
Round

```
table = Table.new()

table = Table.add_player(table, "Péter")
table = Table.add_player(table, "Viktor")
table = Table.add_player(table, "Saša")
```

```
table = Table.start_round(table)

table = Table.hit(table)
table = Table.stay(table)
...
```

```
Table.in_play?(table)

Table.current_player(table)
```

```
def new do
  %Table{
    players: %{},
    round: nil,
    deck: Deck.shuffled()
  }
end
```

```elixir
def add_player(table, player) do
  players = Map.put(table.players, player, 0)
  %Table{table | players: players}
end
```

```elixir
def start_round(table) do
  players = Map.keys(table.players)
  round = Round.new(players, table.deck)
  %Table{table | round: round}
end
```

```
def hit(table) do
  round = Round.hit(table.round)

  if Round.finished?(round) do
    players = inc_scores(table.players, Round.winners(round))
    %Table{table | players: players, deck: round.deck, round: nil}
  else
    %Table{table | round: round}
  end
end
```

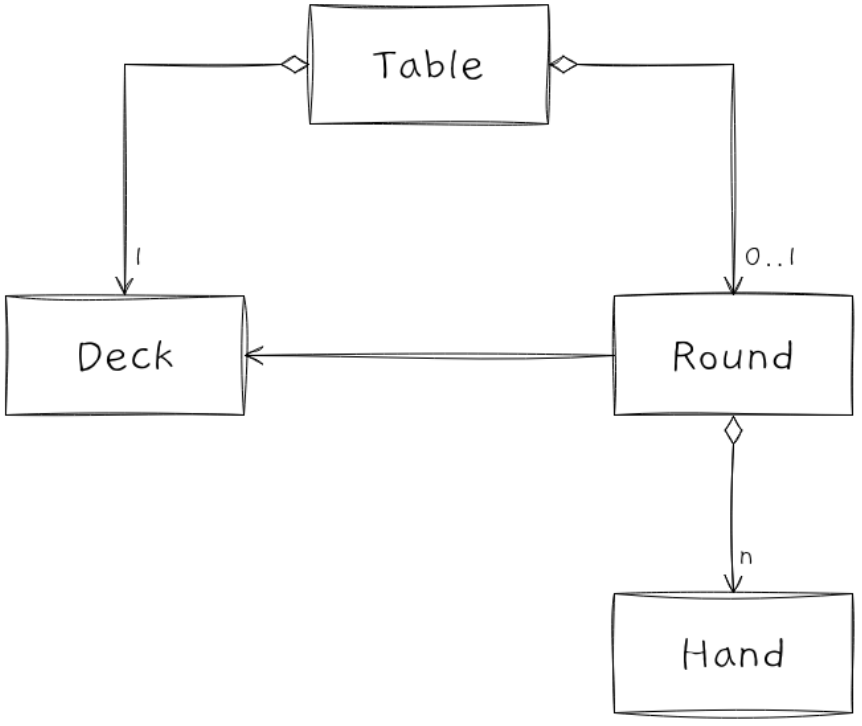


Table
1
0..1
Deck
Round
n
Hand

concurrent Elixir

scalability
responsiveness
fault-tolerance

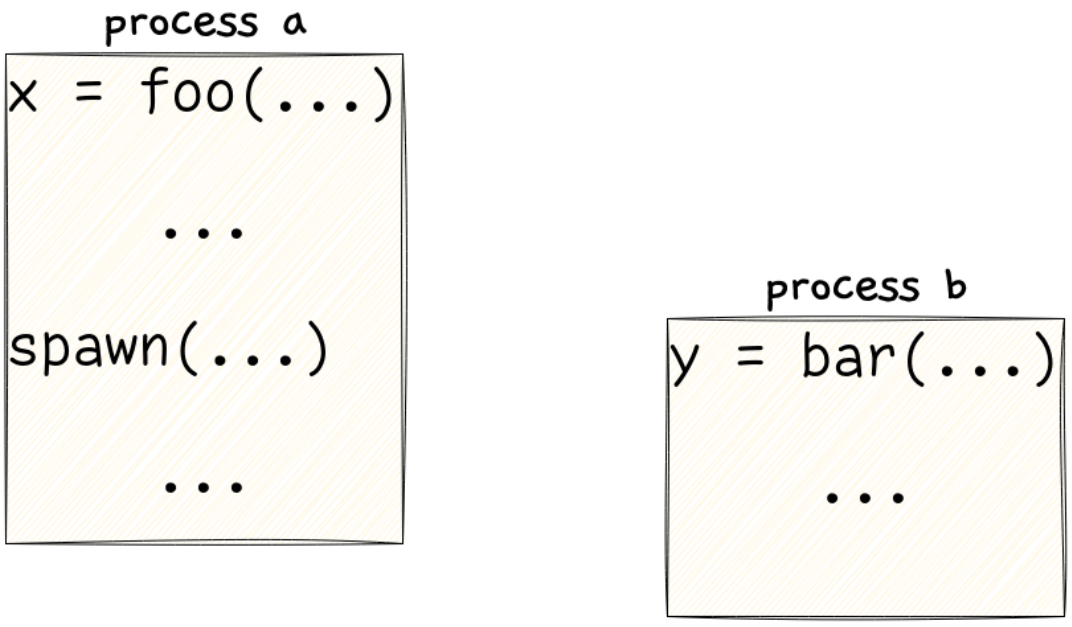
=> high availability

process

not an OS process
sequential program
runtime execution context

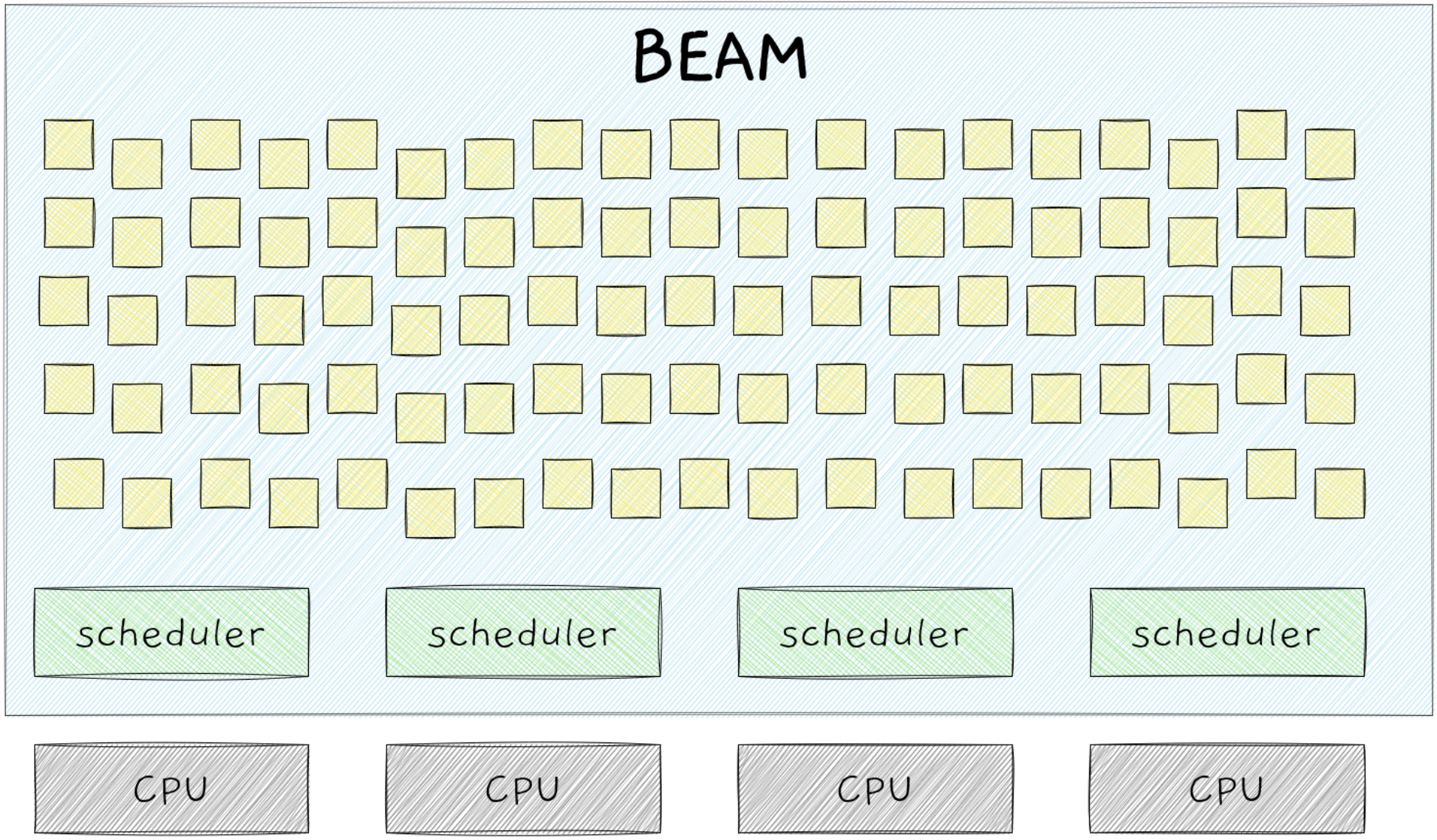
```
deck = Deck.shuffled()
hand = Hand.new()

{card, deck} = Deck.pop()
hand = Hand.add(hand, card)

...
```

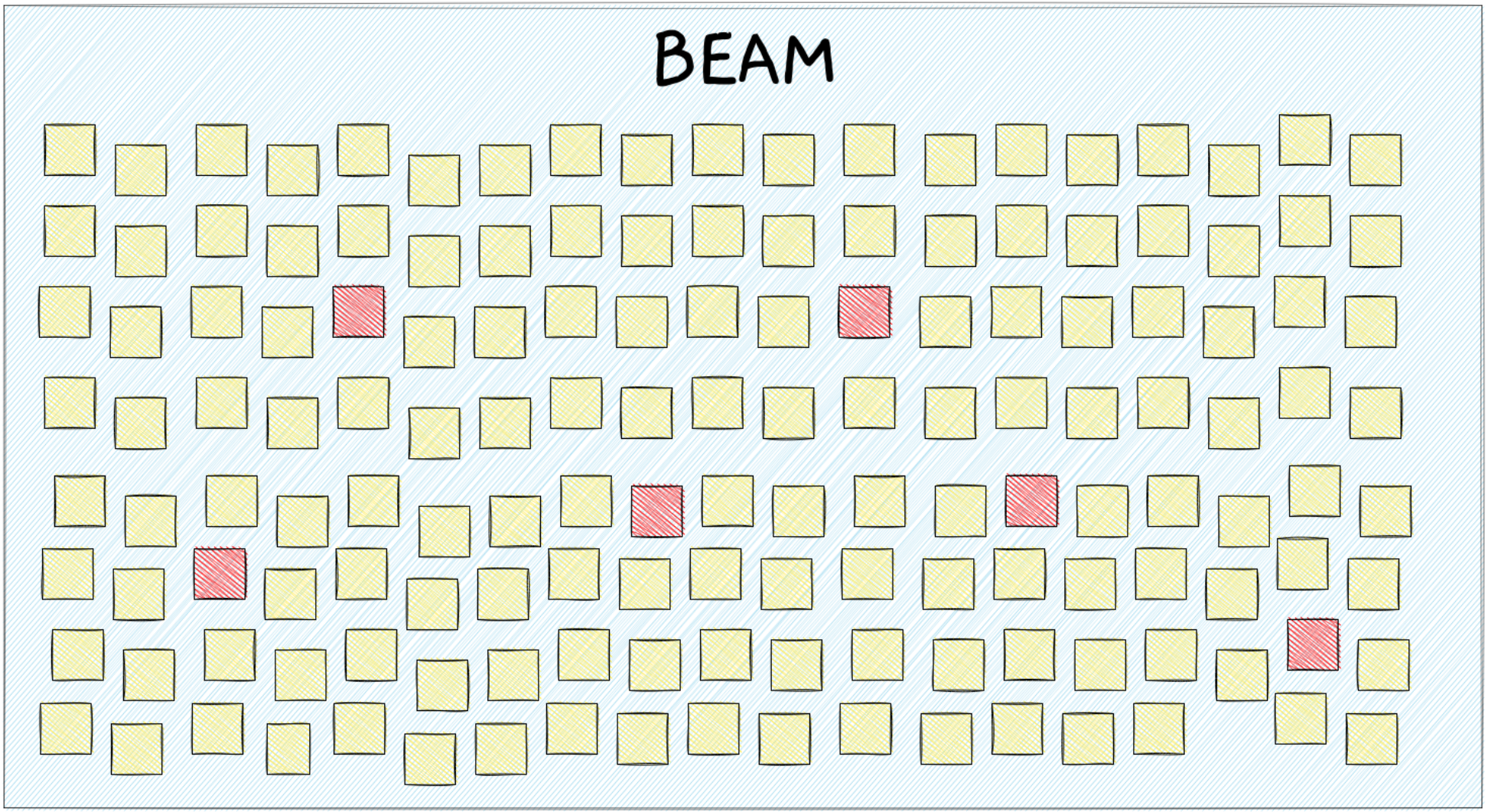
```
x = foo(...)

...

spawn(fn ->
  y = bar(...)
  ...
end)

...
```


process a
x = foo(...)
...
spawn(...)
...
process b
y = bar(...)
...

process

isolated
lightweight

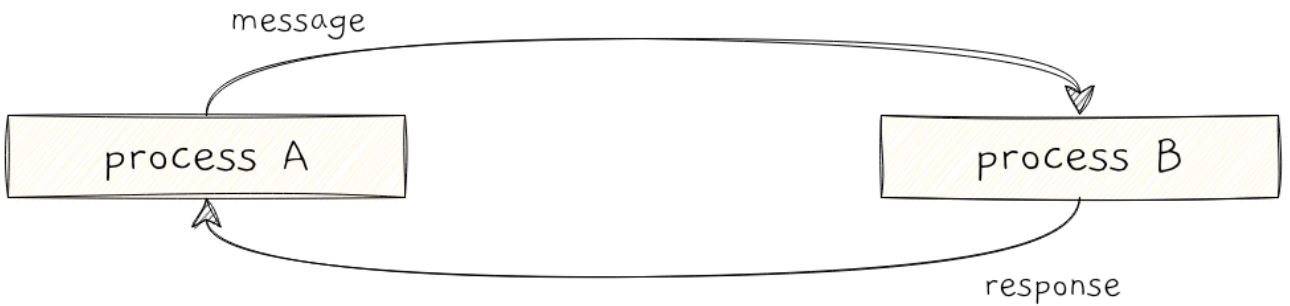

BEAM
scheduler
scheduler
scheduler
scheduler
CPU
CPU
CPU
CPU


BEAM

```
pid = spawn(...)
```

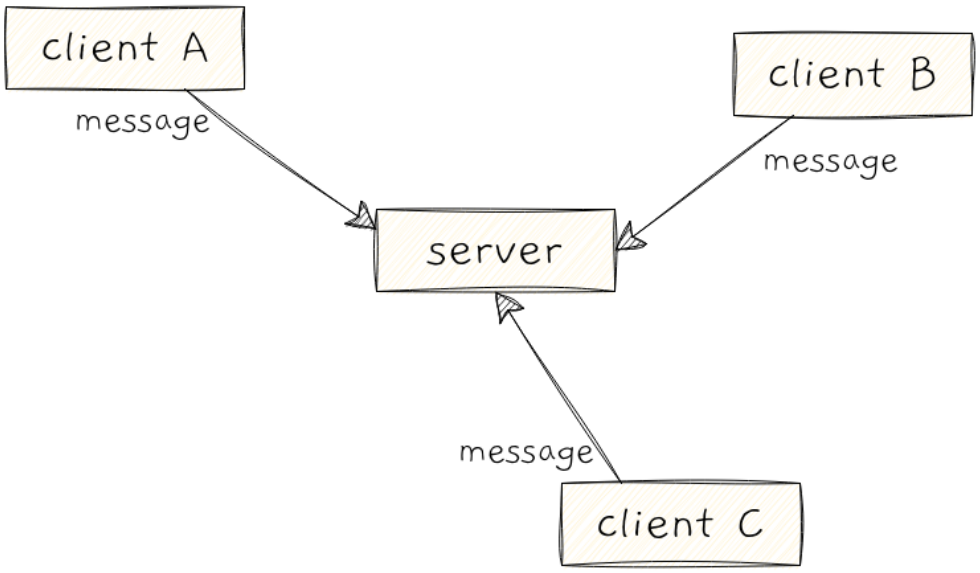
```
message = ...
send(pid, message)
```

```
receive do
  message -> ...
end
```


message
process A
process B
response

```
spawn(fn ->
  initial_state = ...
  loop(initial_state)
end)
```

```
defp loop(state) do
  receive do
    message ->
      state = handle(state, message)
      loop(state)
  end
end
```


client A
client B
message
message
server
message
client C

blackjack site

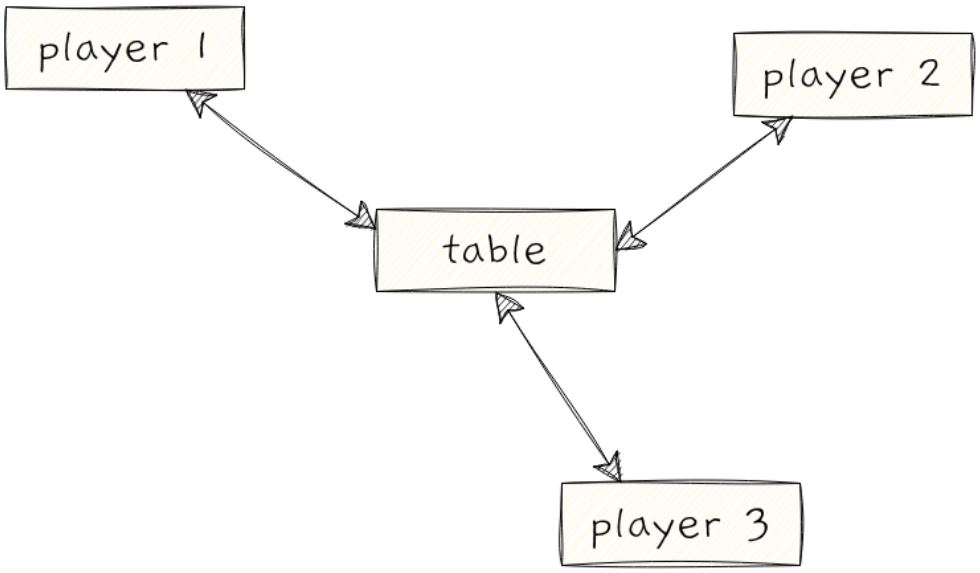
```
pid = TableServer.start()

TableServer.add_player(pid, "Péter")
TableServer.add_player(pid, "Viktor")
TableServer.add_player(pid, "Saša")

TableServer.start_round(pid)

TableServer.hit(pid)
TableServer.stay(pid)
```

```
def start do
  spawn(fn ->
    table = Table.new()
    loop(table)
  end)
end
```

```
defp loop(table) do
  receive do
    message ->
      table = handle_message(message, table)
      loop(table)
  end
end
```

```
def add_player(pid, player),
  do: send(pid, {:add_player, player})
```

```
defp handle_message({:add_player, player}, table),
  do: Table.add_player(table, player)
```


player 1
player 2
table
player 3

Phoenix web framework

one process per connection
one process per request

```elixir
defmodule PlayerSocket do
  def join("player", params, socket) do
    # here we have to join the table
  end

  ...
end
```

```
defmodule PlayerSocket do
  def join("player", params, socket) do
    table = TableRegistry.find_table()
    ...
  end

  ...
end
```

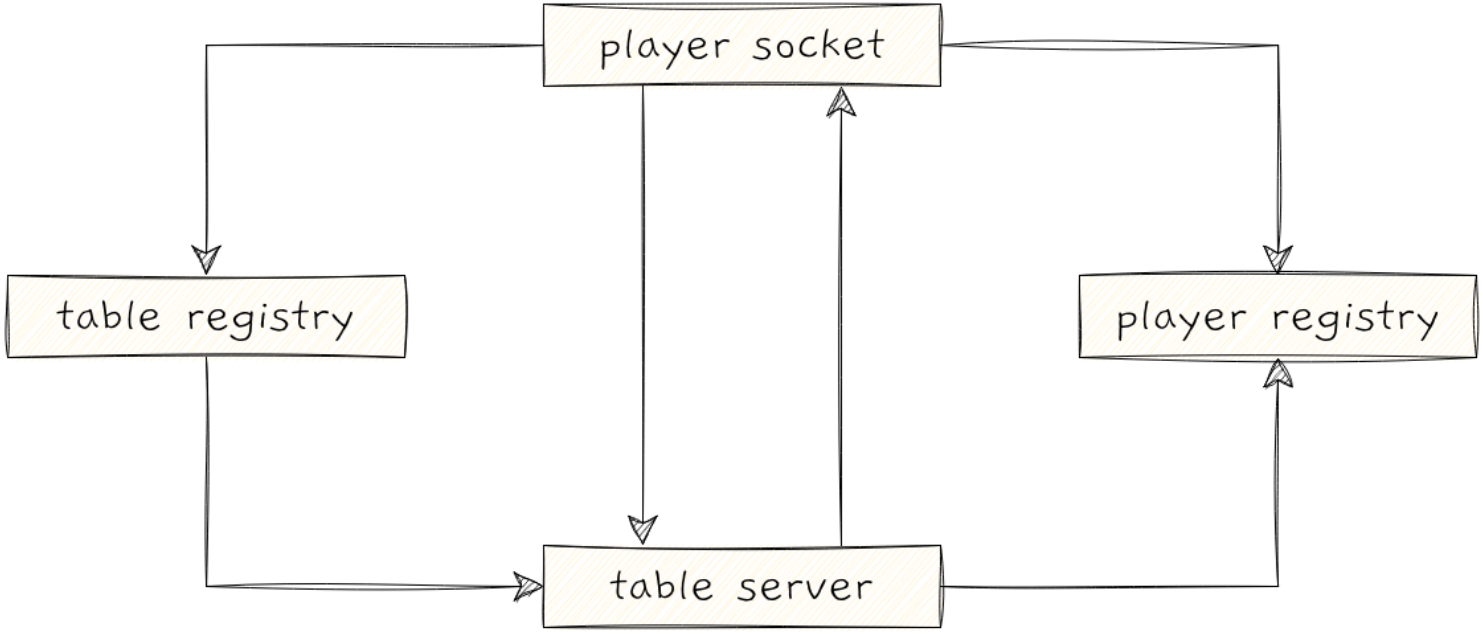
```elixir
# in the table registry
defp handle_message(tables, {:find, caller_pid}) do
  table = Enum.min_by(tables, &Table.num_players/1)

  if table == nil or
       Table.num_players(table) == 4 do
    new_table = Table.start()
    send(caller_pid, {:table, table})
    [new_table | table]
  else
    send(caller_pid, {:table, table})
    tables
  end
end
```

```elixir
# in the player process
def join("player", params, socket) do
  table = TableRegistry.find()
  player = Map.fetch!(params, "player")
  TableServer.add_player(table, player)
  PlayerRegistry.register(self(), table)
  assign(socket, :table, table)
end
```

```elixir
# in the player process
def handle_in("hit", params, socket) do
  table = socket.assigns.table
  TableServer.hit(table)
  ...
end
```

```elixir
# in the table server
defp handle_message(:hit, table) do
  table = Table.hit(table)

  players = PlayerRegistry.players(self())
  Enum.each(players, &send(&1, {:new_table_state, table}))

  table
end
```

```elixir
# in the player process
def handle_info({:new_table_state, table}, socket) do
  push(socket, "new_table_state", table)
  ...
end
```


player socket
table registry
player registry
table server

The Soul of Erlang and Elixir • Sasa Juric • GOTO 2019

Simplifying Systems with Elixir • Sasa Juric • YOW! 2020

BEAM Concurrency in Action • Sasa Juric • YOW! 2022