

Programming Elixir

My Opinions...



Dave Thomas
@pragdave
dave@pragdave.me

Programming

is transforming state in
the presence of inputs

Programming

nested and composed reducers

Programming in Elixir

pattern matching and
trivial reduction functions

```
def get_note_numbers(refs, footnotes, options) do
  Enum.reduce(refs, [], fn(ref, list) →
    case Enum.find(footnotes, &(&1.id == ref)) →
      note = %Block.FnDef{...} → number = length(list) + options.footnote_offset
      %Block.FnDef{ note | number: number }
      list = [note | list]
      list # ToC inline footnotes
    end
  end)
end
```

The Plan

- Pattern Matching
- Managing State
- Processes
- Message Passing
- Agents & Tasks
- GenServer
- Supervisors
- Applications
- Phoenix Channel-based app

The Plan

- You do a lot of coding
- Some exercises
- Mostly incremental development of a Hangman game

Pattern Matching

Pattern Matching

- Match when "assigning", invoking functions, in case functions, exception handlers, ...
- Match on shape and content

Pattern Matching

```
a = 1
```

```
b = [ 1, 2 ]
```

```
[ c, d ] = [ 1, 2 ]
```

```
{ e, f } = { c, d }
```

```
def you_said(1) do  
  "one"  
end
```

```
def you_said(2) do  
  "deux"  
end
```

```
def you_said(n) do  
  "I don't know how to pronounce #{n}"  
end
```

```
you_said 1  => "one"  
you_said 99 => I don't know how to pronounce 99
```

```
defmodule Series do
  def lucas(1), do: 1
  def lucas(2), do: 3
  def lucas(n), do: lucas(n-1) + lucas(n-2)
end
```

```
defmodule Series do
  def lucas(1), do: 1
  def lucas(2), do: 3
  def lucas(n)
    when n > 2 do
      lucas(n-1) + lucas(n-2)
    end
  end
```

More Than Cute

- Pattern matching expresses facts about your code

```
def lucas(1), do: 1
```

this doesn't say "calculate lucas(1) this way"

it says "lucas(1) is 1"

So...

Match, Don't Choose

- A conditional statement implements a choice
- A pattern match *is* the choice

```
def fib(n) do
  if n = 0 do
    1
  else if n = 1 do
    1
  else
    fib(n-1) + fib(n-1)
  end
end
end
```

```
def fib(0), do: 1
def fib(1), do: 1
def fib(n), do: fib(n-1) + fib(n-2)
```

```
def connect(target, verbose) do
  if verbose do
    Logger.info("connecting to #{target}")
  end

  Server.authenticate(target)
  Server.connect(target)

  if verbose do
    Logger.info("connected")
  end
end
```

```
def connect(target, _verbose = true) do
  Logger.info("connecting to #{target}")
  connect(target, false)
  Logger.info("connected")
end

def connect(target, _not_verbose) do
  Server.authenticate(target)
  Server.connect(target)
end
```

```
Enum.reduce(orders, { 0, 0 }, fn order, %{ tax, total } =>
  total = total + order.total
  tax = tax + if tangible?(order) do
    order.total * tax_rate
  else
    0
  end
  { total, tax }
end)
```

```
Enum.reduce(orders, { _tax = 0, _total = 0 }, &get_order_totals/2)

def get_order_totals(order, {tax, total}) do
  {
    tax + tax_for(order, tangible?(order)),
    total + order.total
  }
end

def tax_for(order, _taxable = true), do: order.total * tax_rate
def tax_for(_order, _), do: 0
```

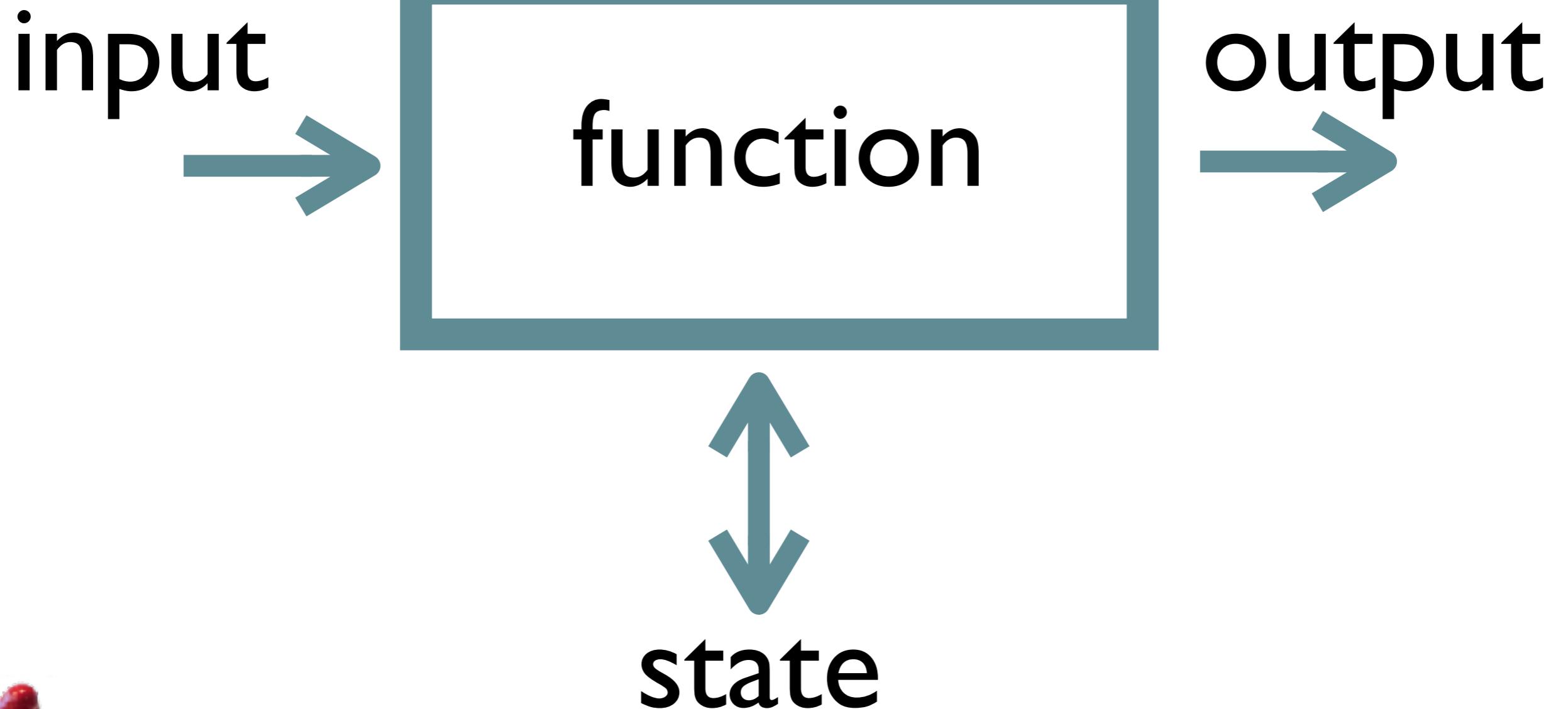
Lab

Time to Tidy

<https://github.com/elixir-lab/lab0>

State in Practice

More than a Convenience



- Function has no state
- It must be given state every time it is invoked
- Only ways are
 - global state
 - pass as parameter (local state)

Global State

- Accessed via well-known name
- Therefore always a singleton
- Represent points of coupling in the app
 - harder to change
 - single point of failure
- Think hard before using

Global State

- Good uses:
 - top-level configuration
 - access to the environment
 - logging
 - registries

Global State

Golden rule:

Convert global state
into local state as soon
as possible.

Local State

- Passed as a parameter
- Functions tend to have the signature

```
func1(state, inputs ... ) → new_state
```

```
func2(state, inputs ... ) → { new_state, result }
```

```
func2(state, inputs ... ) → result
```

Local State

```
func1(state, inputs ... ) → new_state
```

```
func2(state, inputs ... ) → { new_state, result }
```

```
func2(state, inputs ... ) → result
```

"Pure" transformation

Reducer

Local State

```
func1(state, inputs ... ) → new_state
```

```
func2(state, inputs ... ) → { new_state, result }
```

```
func2(state, inputs ... ) → result
```

End of a chain of reducers

Local State

```
func1(state, inputs ... ) → new_state
```

```
func2(state, inputs ... ) → { new_state, result }
```

```
func2(state, inputs ... ) → result
```

Strange uncle no one wants to talk about

Reducers

- Take a state and an input, and return an updated state

```
list = [ 1, 3, 5, 9 ]
```

```
total = Enum.reduce(list, 0, fn (val, sum) → val + sum end)
```

```
IO.puts total
```

input

state

new state

Reducers

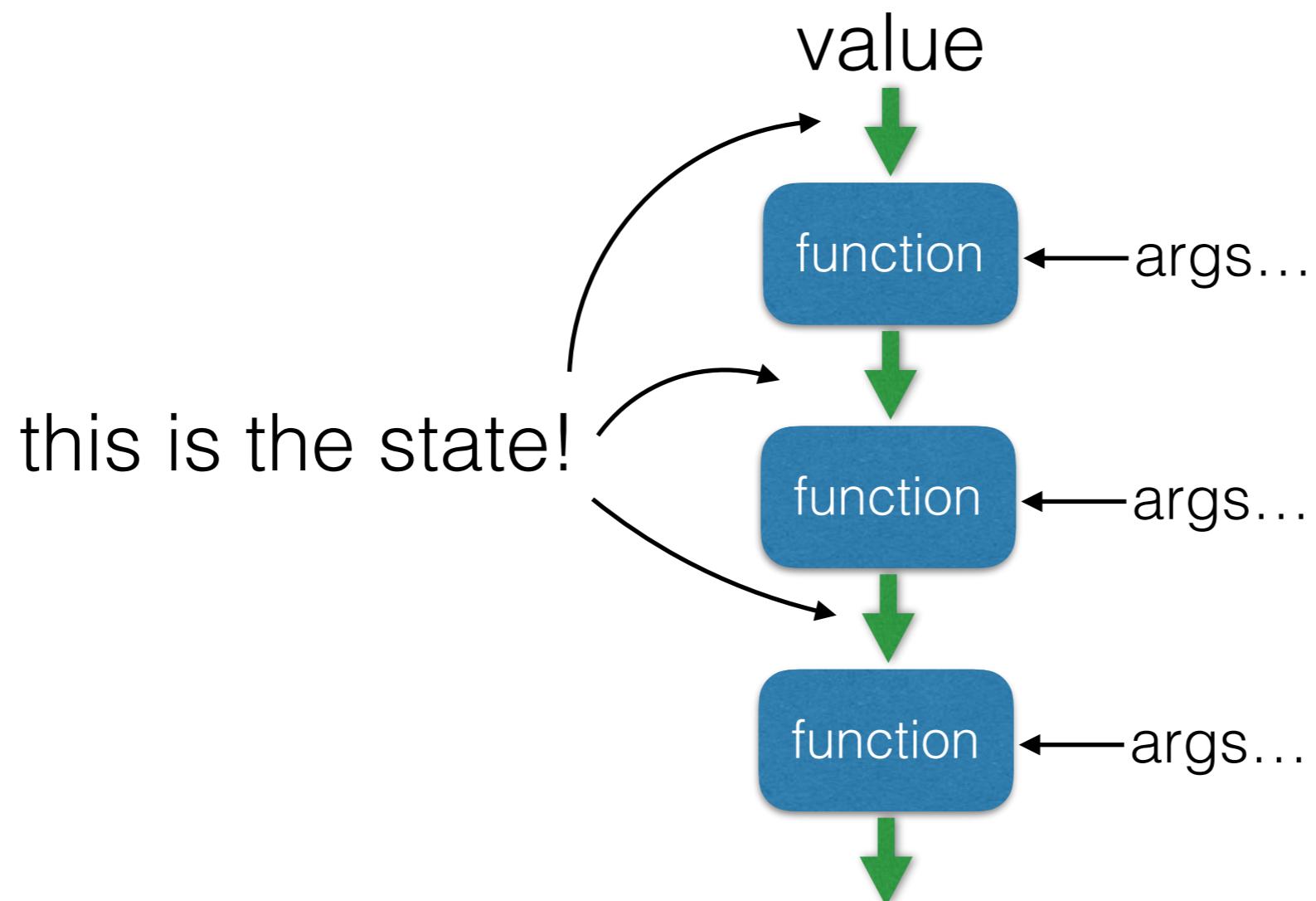
- Take a state and an input, and return an updated state

```
[ 1, 3, 5, 9 ]
```

```
▷ Enum.reduce(0, fn (val, sum) → val + sum end)  
▷ IO.puts
```

- Each step in a pipeline is a reducer!

Pipeline



This is Java/C/Ruby/...

```
list = [ 1, 3, 5, 9 ]
```

```
total = Enum.reduce(list, 0, fn (val, sum) → val + sum end)
```

```
IO.puts total
```

This is nice...

```
[ 1, 3, 5, 9 ]
```

```
▷ Enum.reduce(&(&1+&1))  
▷ IO.puts
```

So...

- Try to write code as pipelines
- Local variables are a smell
- IO.inspect is your friend

But...

```
def deduct(account, amount) do
  if Account.balance(account) < amount do
    { :no_can_do, account }
  else
    account = Account.debit(account, amount)
    account = Account.update_credit_score(account)
    Audit.log(account, :deduct, amount)
    { :ok, account }
  end
end
```

The real world isn't a pipeline...

So make it one

```
def deduct(account, amount) do
  account
    ▷ check_sufficient_funds(amount)
    ▷ maybe_complete_deduction(amount)
end
```

```
def check_sufficient_funds(account, amount) do
  { account, Account.balance(account) < amount }
end
```

```
def maybe_complete_deduction({ account, false }, _amount) do
  { :no_can_do, account }
end
```

```
def maybe_complete_deduction({ account, true }, amount) do
  account
    ▷ Account.debit(amount)
    ▷ Account.update_credit_score()
    ▷ Audit.log(:deduct, amount)
    ▷ ok
end
```

```
def ok(value) do
  { :ok, value }
end
```

```
def deduct(account, amount) do
  if Account.balance(account) < amount do
    { :no_can_do, account }
  else
    account = Account.debit(account, amount)
    account = Account.update_credit_score(account)
    Audit.log(account, :deduct, amount)
    { :ok, account }
  end
end
```

So...

- Pattern matching trumps conditionals
 - makes code more modular. Easier to read, test, and change
- Add stuff to your state as you progress through the pipeline
- Don't destroy your code doing this. `cond` and `case` are ok in moderation
- (never let Dave write an accounting system)

Lab

Hangman using pattern matching

git clone <https://github.com/elixir-lab/lab2>

Processes

The rich-person's objects

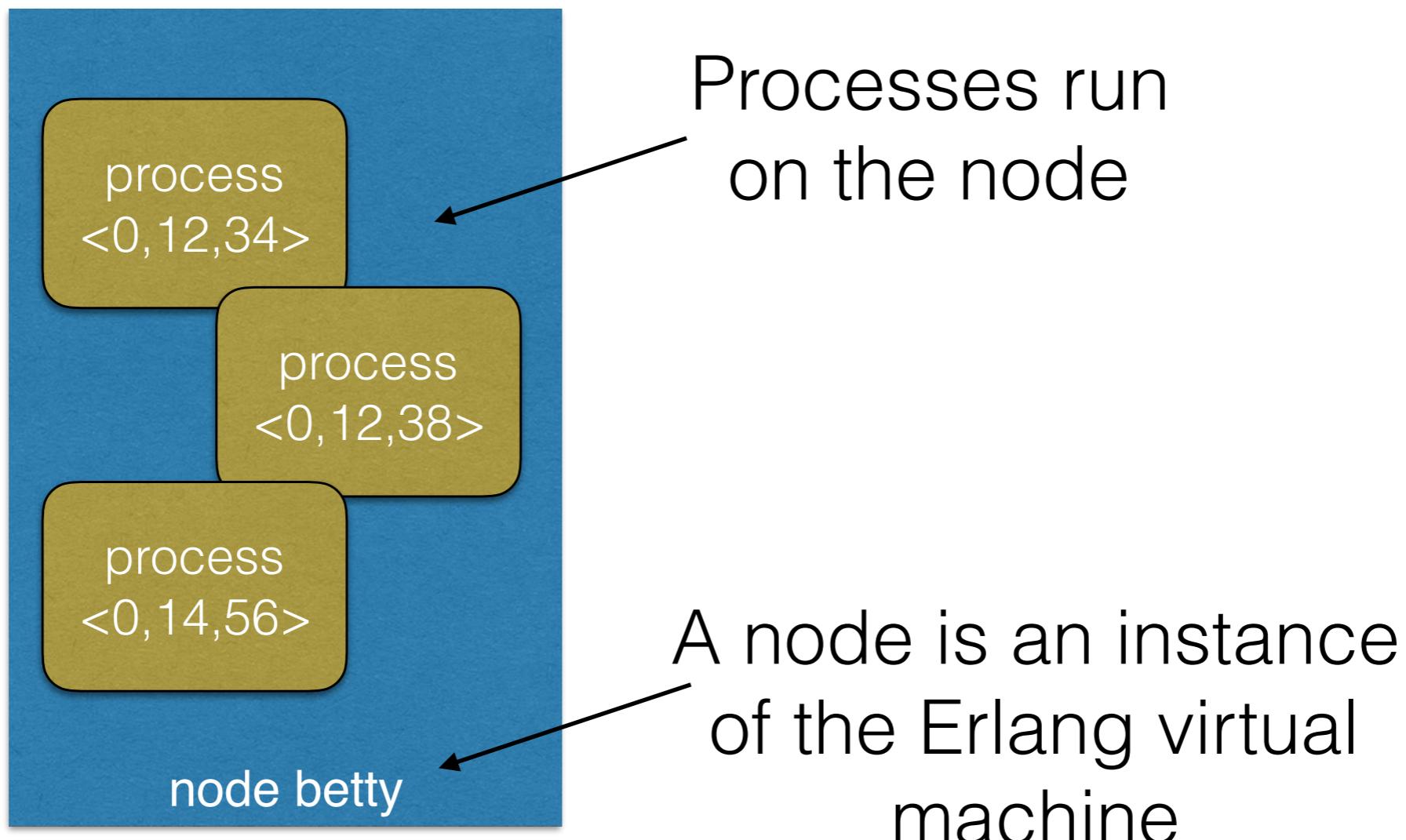
Why Processes?

- Practical reasons
 - concurrency / parallelism
 - faster GC
 - better error isolation
 - inherently distributed and scalable
 - monitoring and management

Why Processes?

- Design reasons
 - act as state holders
 - pure isolation
 - totally decoupled
 - OO done right

Erlang Model



Processes

- Not OS processes, threads, or fibers.
- Implemented inside the Erlang VM, which has its own scheduler, memory manager, etc. Basically it's its own OS

Why?

- Performance
 - initial memory usage of a new process is just over 1kb.
Most of this is stack and heap.
 - processes take less than a microsecond to create
 - sending messages between processes is typically sub-microsecond
 - GC is more efficient

Why?

- **Resilience**
 - the VM knows what a process's state means, and can intelligently handle errors
 - the VM can arrange for processes to monitor each other
 - the VM can monitor the internals of processes and detect potential problems early

How?

- the VM runs (by default) one operating system thread per available CPU.
- each thread has a process scheduler, and these schedulers cooperate to keep the CPUs busy.
- process switching is done:
 - when a process has to wait for a message
 - after n functions have been called

Creating a Process

anonymous function

```
spawn(fn →  
  IO.puts("In the child, pid = #{inspect self()}")  
end)
```

named function

```
defmodule DoSomething do  
  def expensive(x, y) do  
    # some really long calculation  
    IO.puts "based on #{x} and #{y}, the answer is 42"  
  end  
end
```

```
spawn(DoSomething, :expensive, [ 12, 34 ])
```

Creating a Process

```
spawn(fn →  
  I0.puts("In the child, pid = #{inspect self()}")
end)
```

anonymous function

defmodule DoSomething do
 def expensive(x,y) **do**
 # some really long calculation
 I0.puts"based on #{x} and #{y}, the answer is 42"
 end
end

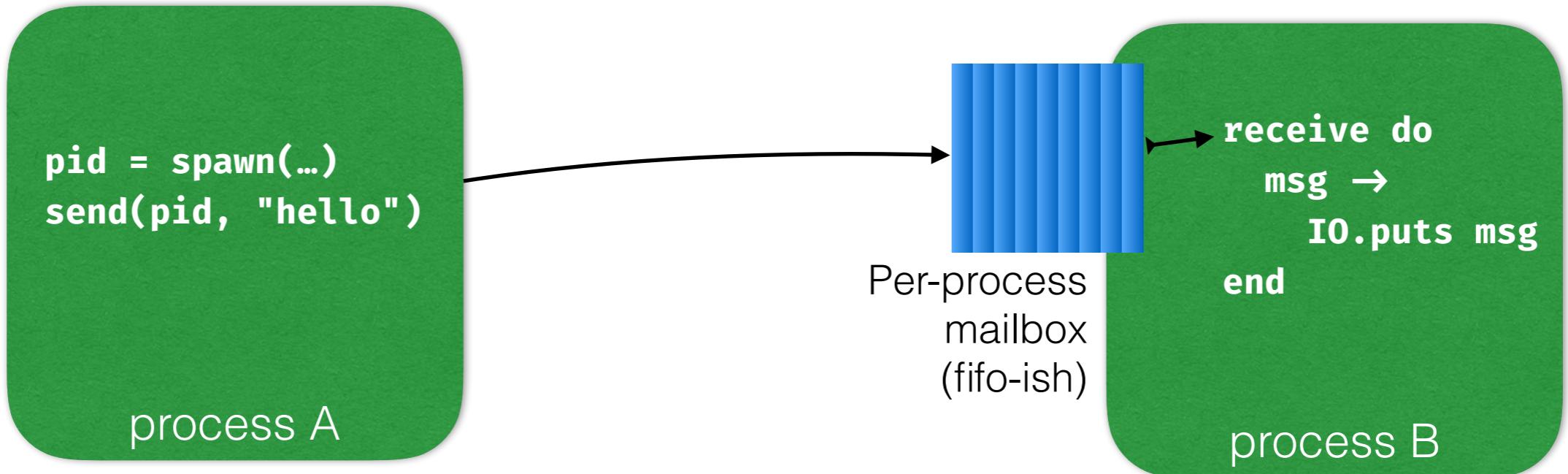
named function

```
spawn(DoSomething, :expensive, [ 12, 34 ])
```

**Too low-level for
99% of code**

Message Passing

The Heartbeat of Your App



You get

- fast messages
- async delivery
- selective receives
- order guaranteed between sender and receiver

You don't get

- message priorities
- two-way messages (use two one-way)
- order guarantees between two senders and receiver

```
receive do
    pattern_1 →
        # handle message

    pattern_2 when «guard» →
        # handle message

    after «timeout» →
        # handle timeout

end
```

```
defmodule Receiver do
  def listen(special) do
    receive do
      ^special →
        IO.puts "you're so special"
      other →
        IO.puts "#{other} is average"
    after 10000 →
      IO.puts "I'm bored"
    end
  end
end
```

```
pid = spawn(Receiver, :listen, [ "barney" ])
send(pid, "barney") # ⇒ you're so special
```

```
pid = spawn(Receiver, :listen, [ "barney" ])
send(pid, "fred") # ⇒ fred is average
```

```
pid = spawn(Receiver, :listen, [ "barney" ])  
  
send(pid, "fred")  # ⇒ fred is average  
  
send(pid, "barney") # ⇒ *crickets*
```

```
def listen(special) do  
  receive do  
    ^special →  
      IO.puts "you're so special"  
    other →  
      IO.puts "#{other} is average"  
  after 10000 →  
    IO.puts "I'm bored"  
  end  
end
```

Function exits
after handling
the message

```
def listen(special) do
  receive do
    ^special →
      IO.puts "you're so special"
    other →
      IO.puts "#{other} is average"
  after 10000 →
    IO.puts "I'm bored"
  end
  listen(special)
end
```

So let's
loop

```
pid = spawn(Receiver, :listen, [ "barney" ])

send(pid, "fred")    # ⇒ fred is average

send(pid, "barney") # ⇒ you're so special

send(pid, "dino")   # ⇒ dino is average
```

```
def listen(special) do
  receive do
    # do stuff...
  end
  listen(special)
end
```

special is
passed
between
receipts of
messages

```
def total(n \\ 0) do
  receive do
    {:add, value} when is_integer(value) →
      IO.inspect [n, value]
      total(n+value)
    :done →
      IO.puts "Total is #{n}"
  end
end
```

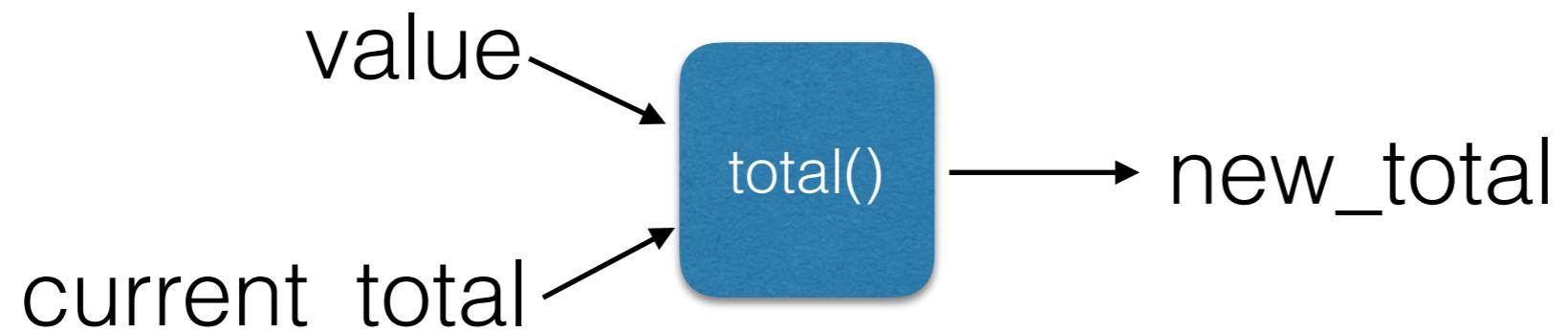
```
iex> pid = spawn Receiver, :total, []
#PID<0.110.0>
```

```
iex> send pid, {:add, 2}
[0, 2]
{:add, 2}
```

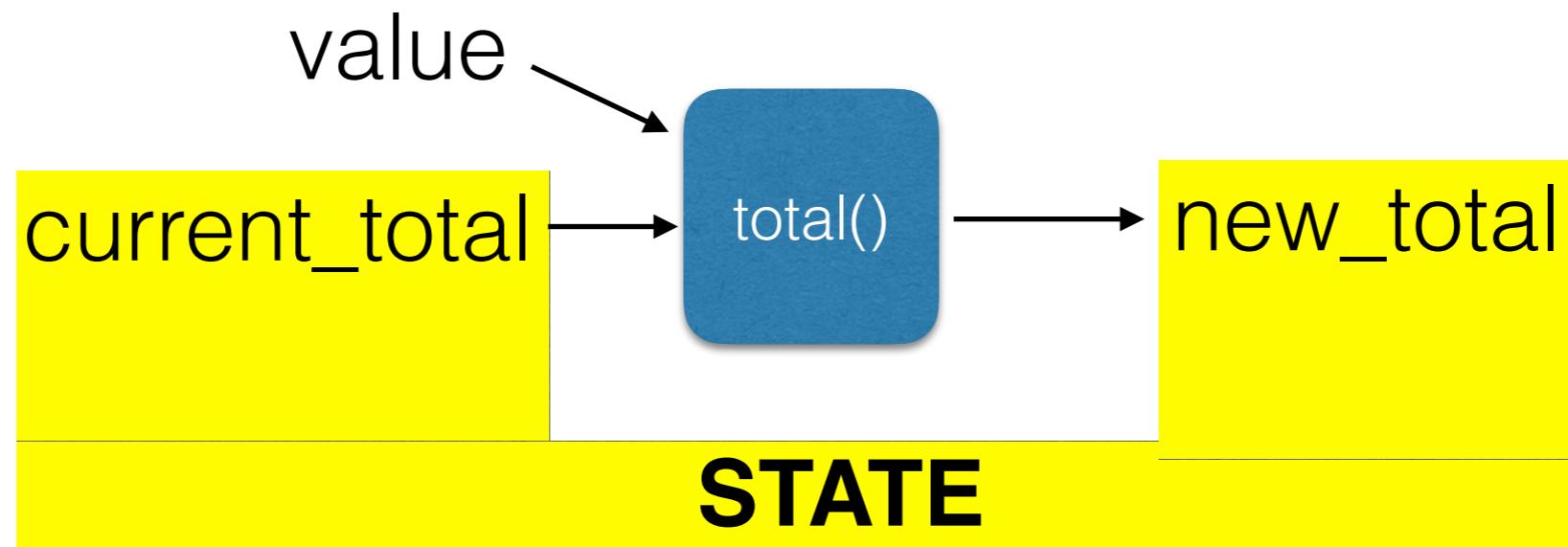
```
iex> send pid, {:add, 2}
[2, 2]
{:add, 2}
```

```
iex> send pid, :done
Total is 4
:done
```

```
def total(n \\ 0) do
  receive do
    {:add, value} when is_integer(value) →
      IO.inspect [n, value]
      total(n+value)
    :done →
      IO.puts "Total is #{n}"
  end
end
```



Our Message Loop is a Reducer!



Lab

<https://github.com/elixir-lab/lab3>

Playing with Processes

Higher Level Abstractions

- Agents
- Tasks
- OTP GenServers

Agent

- Place to store state
- Pass it a function, and it applies the function to the state and either
 - returns the result
 - updates the state with the result
 - updates the state *and* returns a result (atomically)

```
iex> {:ok, pid} = Agent.start_link(fn → 0 end)  
{:ok, #PID<0.84.0>}
```

```
iex> Agent.update(pid, fn total → total + 3 end)  
:ok
```

```
iex> Agent.update(pid, fn total → total + 2 end)  
:ok
```

```
iex> Agent.get(pid, fn total → "The total is #{total}" end)  
"The total is 5"
```

Wrap agents to give them an API specific their particular use

```
defmodule Totaller do
  def init(start_value \\ 0) do
    { :ok, agent } = Agent.start_link(fn → start_value end)
    agent
  end

  def add(agent, value) do
    Agent.update(agent, fn total → total + value end)
  end

  def current_value(agent) do
    Agent.get(agent, fn total → "The total is #{total}" end)
  end
end
```

```
defmodule Totaller do
  def init(start_value \\ 0) do
    { :ok, agent } = Agent.start_link(fn → start_value end)
    agent
  end

  def add(agent, value) do
    Agent.update(agent, fn total → total + value end)
  end

  def current_value(agent) do
    Agent.get(agent, fn total → "The total is #{total}" end)
  end
end
```

```
iex> sum = Totaller.init(123)
#PID<0.128.0>
```

```
iex> Totaller.add(sum, 55)
:ok
```

```
iex> Totaller.current_value(sum)
"The total is 178"
```

```
iex> Totaller.add(sum, 12)
:ok
```

```
iex> Totaller.current_value(sum)
"The total is 190"
```

Task

- Normally used to run some code in the background, and later to wait for the result to become available
- (Like promises or futures in some other languages)

```
twitter_task = Task.async(fn → get_twitter_feed(user) end)
facebook_task = Task.async(fn → get_facebook_timeline(user) end)

twitter_feed      = Task.await(twitter_task)
facebook_timeline = Task.await(facebook_task)
```

- The app will fetch from Twitter and Facebook in parallel
- It doesn't matter which finishes first
- Overall time taken by this code is the time taken by the slowest fetch (plus a little...)

Named Processes

Give Agents & Tasks Names

```
Agent.start_link(fn → %{} end, name: Memory)  
  
for { k, v } ← [ fred: 12, wilma: 23, betty: 34, barney: 45 ] do  
  Agent.update(Memory, fn map → Map.put(map, k, v) end)  
end  
  
Agent.get(Memory, fn map → map[:fred] end)
```

Use the name
in place of the pid

Just an atom...

Give Spawned Processes Names

```
iex> pid = spawn(fn →  
... >   receive do  
... >     something → IO.puts "you said #{something}"  
... >   end  
... > end)  
#PID<0.116.0>
```

Associate name
with pid

```
iex> Process.register(pid, Echo)  
true
```

```
iex> send Echo, "hello"  
you said hello
```

Use the name
in place of the pid

OTP GenServer etc...

All in good time, my little pretties.

All in good time.

Lab

Playing with Agents —a Better Dictionary

<https://github.com/elixir-lab/lab4>

GenServer

Consistent Servers

```
def total(n \\ 0) do
  receive do
    {:add, value} when is_integer(value) →
      IO.inspect [n, value]
      total(n+value)
    :done →
      IO.puts "Total is #{n}"
  end
end
```

```
def total(n \\ 0) do
  receive do
    {:add, value} when is_integer(value) →
      IO.inspect [n, value]
      total(n+value)
    :done →
      IO.puts "Total is #{n}"
  end
end

def handle_messages(module, state) do
  receive do
    msg →
      { status, result, new_state } = apply(module, :handle, [ msg, state ])
      if status ≠ :stop do
        handle_messages(module, state)
      else
        result
      end
    end
  end

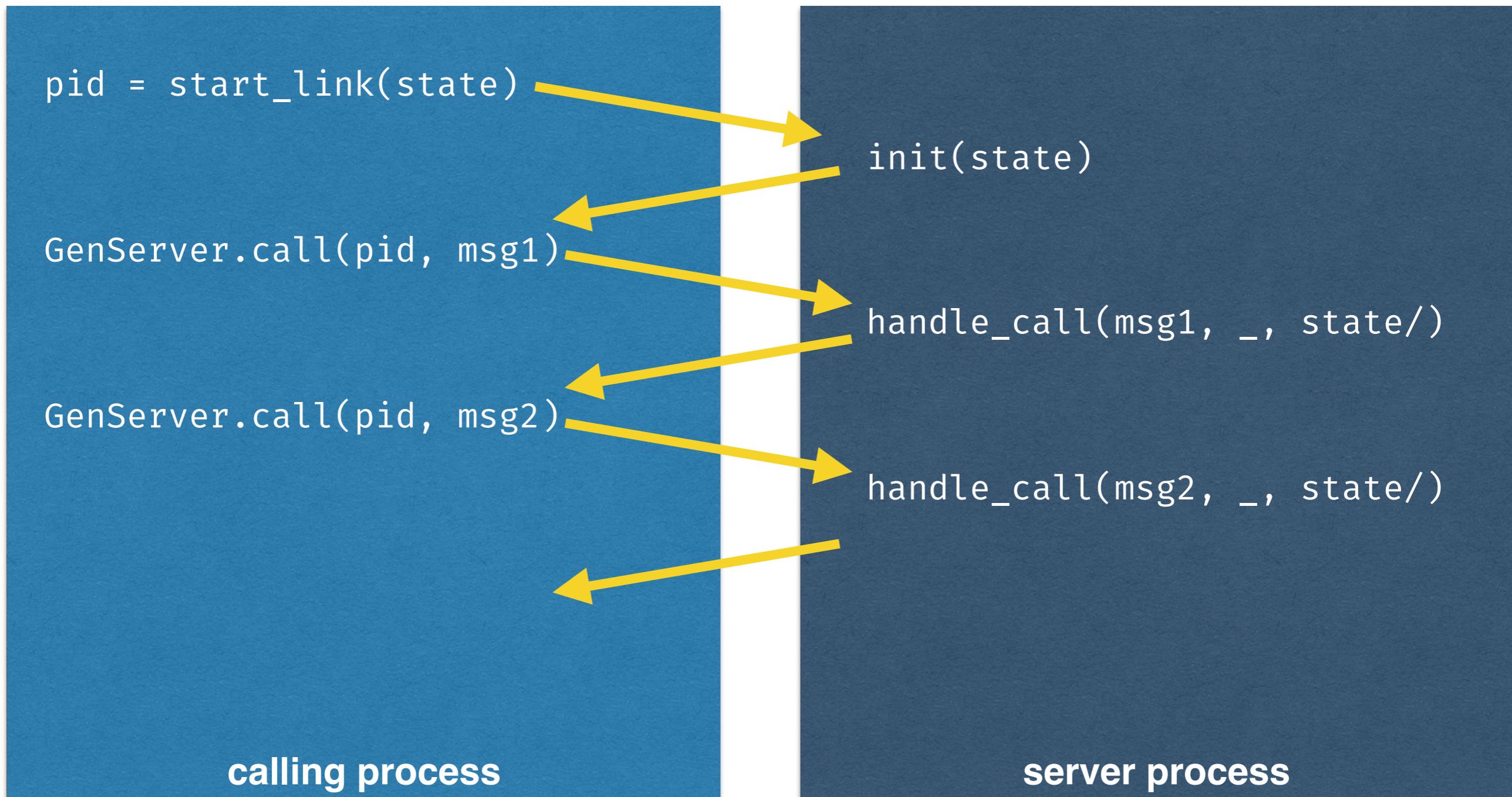
defmodule Summer do
  def handle({:add, n}, total_so_far) do
    { :ok, n, total_so_far + n }
  end

  def handle({:done}, total_so_far) do
    { :stop, total_so_far, nil }
  end
end
```

GenServer

- Is a utility module that lets you write server processes without worrying about the low level details
- Way more complex than the previous slide—handles timeouts, control messages, process failure, logging, ...

Lifecycle



Lifecycle

```
def start_link(initial_value \\ 0) do
  GenServer.start_link(<<some module>>,
    initial_value)
end

def add(pid, value) do
  GenServer.call(pid, {:add, value})
end

def done(pid) do
  GenServer.call(pid, {:done})
end
```

calling process

```
def init([initial_value]) do
  state = %{ total: initial_value,
             call_count: 0 }
  { :ok, state }
end

def handle_call({:add, value}, _, state) do
  new_total      = state.total + value
  new_call_count = state.call_count + 1
  new_state = %{ total: new_total,
                 call_count: new_call_count }

  { :reply, new_total, new_state }
end

def handle_call({:done}, _, state) do
  { :reply,
    "called #{state.call_count} times.
    total=#{state.total}", state }
end
```

server process

```
defmodule Summer do
  use GenServer
```

```
  def start_link(initial_value \\ 0) do
    GenServer.start_link(Summer, [initial_value])
  end
```

```
  def add(pid, value) do
    GenServer.call(pid, {:add, value})
  end
```

```
  def done(pid) do
    GenServer.call(pid, {:done})
  end
```

API

```
def init([initial_value]) do
  state = %{ total: initial_value, call_count: 0 }
  { :ok, state }
end
```

```
def handle_call({:add, value}, _, state) do
  new_total      = state.total + value
  new_call_count = state.call_count + 1
  { :reply, new_total, %{ total: new_total, call_count: new_call_count } }
end
```

```
def handle_call({:done}, _, state) do
  { :reply, "called #{state.call_count} times. total=#{state.total}", state }
end
```

Implementation
(in server process)

end

```
def init([initial_value]) do
  state = %{ total: initial_value, call_count: 0 }
  { :ok, state }
end
```

`init` receives the value passed to `GenServer.start_link` and uses it to create the process state. This hides the implementation of the state from the outside world.

```
def handle_call({:add, value}, _, state) do
  new_total      = state.total + value
  new_call_count = state.call_count + 1
  { :reply, new_total, %{ total: new_total, call_count: new_call_count } }
end
```

`handle_call` pattern matches on the message given in `GenServer.call`. The second parameter, the pid of the sender, is only used if the server wants to respond before completing its work. The last parameter is the state.

`handle_call` returns `:reply`, the actual value to be given to the caller, and the (potentially updated) state

```
defmodule Summer do
  use GenServer
```

```
  def start_link(initial_value \\ 0) do
    GenServer.start_link(Summer, [initial_value])
  end

  def add(pid, value) do
    GenServer.call(pid, {:add, value})
  end

  def done(pid) do
    GenServer.call(pid, {:done})
  end
```

```
  def init([initial_value]) do
    state = %Summer{initial_value: initial_value, call_count: 0}
    { :ok, state }
  end

  def handle_call({:add, value}, _, state) do
    new_total = state.total + value
    new_call_count = state.call_count + 1
    { :reply, new_total, %{ total: new_total, call_count: new_call_count } }
  end

  def handle_call({:done}, _, state) do
    { :reply, "called #{state.call_count} times. total=#{state.total}", state }
  end

end
```

Multiple Concerns

- API vs Application logic
- Caller process vs server process
- Housekeeping vs "the meat"

Fix #1: Delegate

- Write all your application-specific logic in its own module. No server stuff.
- Write a GenServer that delegates to this module
- Benefits:
 - you can test your application logic directly
 - you can start without a server, then add it later

```
defmodule Summer do
  use GenServer

  def start_link(initial_value \\ 0) do
    GenServer.start_link(Summer, [initial_value])
  end

  def add(pid, value) do
    GenServer.call(pid, {:add, value})
  end

  def done(pid) do
    GenServer.call(pid, {:done})
  end

  def init([initial_value]) do
    state = Summer.Implementation.init(initial_value)
    { :ok, state }
  end

  def handle_call({:add, value}, _, state) do
    result = Summer.Implementation.add(state, value)
    { :reply, result, result }
  end

  def handle_call({:done}, _, state) do
    { :reply, Summer.Implementation.done(state), state }
  end
end
```

API

Server

```
defmodule Summer.Implementation do
  def init(total), do: total

  def add(total, value), do: total + value

  def done(total), do: "The total is: " ++ Integer.to_string(total)

end
```

Implementation

Lab

Make the game into a server

<https://github.com/elixir-lab/lab5>

Better, but...

Can anyone spot
any duplication?

```
defmodule Summer do
  use GenServer

  def start_link(initial_value \\ 0) do
    GenServer.start_link(Summer, [initial_value])
  end

  def add(pid, value) do
    GenServer.call(pid, {:add, value})
  end

  def done(pid) do
    GenServer.call(pid, {:done})
  end

  def init([initial_value]) do
    state = Summer.Implementation.init(initial_value)
    { :ok, state }
  end

  def handle_call({:add, value}, _, state) do
    result = Summer.Implementation.add(state, value)
    { :reply, result, result }
  end

  def handle_call({:done}, _, state) do
    { :reply, Summer.Implementation.done(state), state }
  end

defmodule Summer.Implementation do
  def init(total), do: total
  def add(total, value), do: total + value
  def done(total), do: "The total is #{total}"
end
```

Better, but...

Can anyone spot
any ~~duplication~~
triplication?

```
defmodule Summer do
  use GenServer

  def start_link(initial_value \\ 0) do
    GenServer.start_link(Summer, [initial_value])
  end

  def add(pid, value) do
    GenServer.call(pid, {:add, value})
  end

  def done(pid) do
    GenServer.call(pid, {:done})
  end

  def init([initial_value]) do
    state = Summer.Implementation.init(initial_value)
    { :ok, state }
  end

  def handle_call({:add, value}, _, state) do
    result = Summer.Implementation.add(state, value)
    { :reply, result, result }
  end

  def handle_call({:done}, _, state) do
    { :reply, Summer.Implementation.done(state), state }
  end

defmodule Summer.Implementation do
  def init(total), do: total
  def add(total, value), do: total + value
  def done(total), do: "The total is #{total}"
end
```

Jeeves

```
defmodule Summer.Implementation do  
  
  def init(total), do: total  
  
  def add(total, value), do: total + value  
  
  def done(total), do: "The total is #{total}"  
  
end
```

Jeeves

```
defmodule Summer do
  use Jeeves.Anonymous, initial_state: 0

  def add(total, value) do
    set_state(total + value)
    value
  end
end

def done(total), do: "The total is #{total}"

end
```

Jeeves

```
defmodule Summer do
  use Jeeves.Anonymous, initial_state: 0, show_code: true

  def add(total, value) do
    set_state(total + value) do
      value
    end
  end

  def done(total), do: "The total is #{total}"
end
```

```
defmodule Summer do
  import(Kernel, except: [def: 2])
  import(Jeeves.Common, only: [def: 2, set_state: 1, set_state: 2])
  @before_compile({Jeeves.Anonymous, :generate_code_callback})
  def run() do
    run(:no_state)
  end
  def run(state_override) do
    state = initial_state(state_override, :no_state)
    {:ok, pid} = GenServer.start_link(__MODULE__, state, server_opts())
    pid
  end
  def init(state) do
    {:ok, state}
  end
  def initial_state(override, _default) do
    override
  end
  def server_opts() do
    []
  end
  defoverridable(initial_state: 2, init: 1)

  use(GenServer)
  def add(total, value) do
    GenServer.call(total, {:add, value})
  end
  def done(total) do
    GenServer.call(total, {:done})
  end
  def handle_call({:add, value}, _, total) do
    __MODULE__.Implementation.add(total, value) ▷ Jeeves.Common.create_g
    response(total)
  end
  def handle_call({:done}, _, total) do
    __MODULE__.Implementation.done(total) ▷ Jeeves.Common.create_g
    response(total)
  end
  defmodule(Implementation) do
    def add(total, value) do
      set_state(total + value) do
        value
      end
    end
    def done(total) do
      "The total is #{total}"
    end
  end
end
```

Configuration

API

Server

Implementation

Jeeves

```
defmodule Summon do
  use Jeeves.Anonymous, initial_state: 0

  def add(total, value) do
    set_state(total + value)
    value
  end
end

def done(total), do: "The total is #{total}"

end
```

Named Server

```
defmodule Summer do
  use Jeeves.Named, initial_state: 0, state_name: :total

  def add(value) do
    set_state(total + value) do
      value
    end
  end

  def done(), do: "The total is #{total}"
end
```

Named Server

```
defmodule Summer do
  use Jeeves.Named, initial_state: 0, state_name: :total

  def add(value) do
    set_state(total + value) do
      value
    end
  end

  def done(), do: "The total is #{total}"
end
```

```
iex> Summer.run 0
#PID<0.339.0>
```

```
iex> Summer.add 1
1
```

```
iex> Summer.add 5
5
```

```
iex> Summer.done
"The total is 6"
```

Supervision

The Crash Stops Here

Designing for Failure

- You cannot stop things failing
- You can minimize the impact of those failures
- You can localize the damage done by those failures

`start` and `start_link`

- `start` creates a totally independent process.
- `start_link` creates a dependent process:
 - if it crashes, you crash
 - if you crash, it crashes

`start` and `start_link`

- In general, you want to use `start_link` as this will prevent errors happening without you knowing about it.
- But... this is mutually assured destruction.

Monitoring

- you can *monitor* a process
- the monitoring process gets notified if the monitored process exits
- this lets you handle errors
- but it is very, very difficult to get right

Supervision

- Wrapper around monitoring, just like GenServer is a wrapper around spawning processes
- Supervisor monitors a set of subprocesses
 - workers (GenServers)
 - subsupervisors

Supervision

- Starts subprocesses in an orderly manner
- Monitors them, and handles failure

On failure

- Restart that worker
- Restart all workers
- Restart the failed worker and all workers after it in the child list
- Give up and fail the supervisor

External Supervisor

```
defmodule Summer.Supervisor do
  use Supervisor

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

  def init([]) do
    children = [
      worker(Summer, [0]),
      worker(Divider, [1]),
    ]
    supervise(children, strategy: :one_for_one)
  end
end
```

Inline Supervisor

```
def start do
  import Supervisor.Spec

  children = [
    worker(__MODULE__, [])
  ]

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

Often used in top-level applications

Server

Supervisor

```
defmodule MyServer do

  use GenServer

  def start do
    import Supervisor.Spec

    children = [
      worker(__MODULE__, [])
    ]

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

  def start_link do
    GenServer.start_link(__MODULE__, 0, name: Buggy)
  end

  def init(state) do
    IO.puts "server pid is #{inspect self()}"
    { :ok, state }
  end

  def good_call(), do: GenServer.call(Buggy, :good)
  def crash_me(), do: GenServer.call(Buggy, :crash_me)

  def handle_call(:good, _, state), do: { :reply, state, state+1 }
  def handle_call(:crash_me, _, state), do: state / 0
end
```

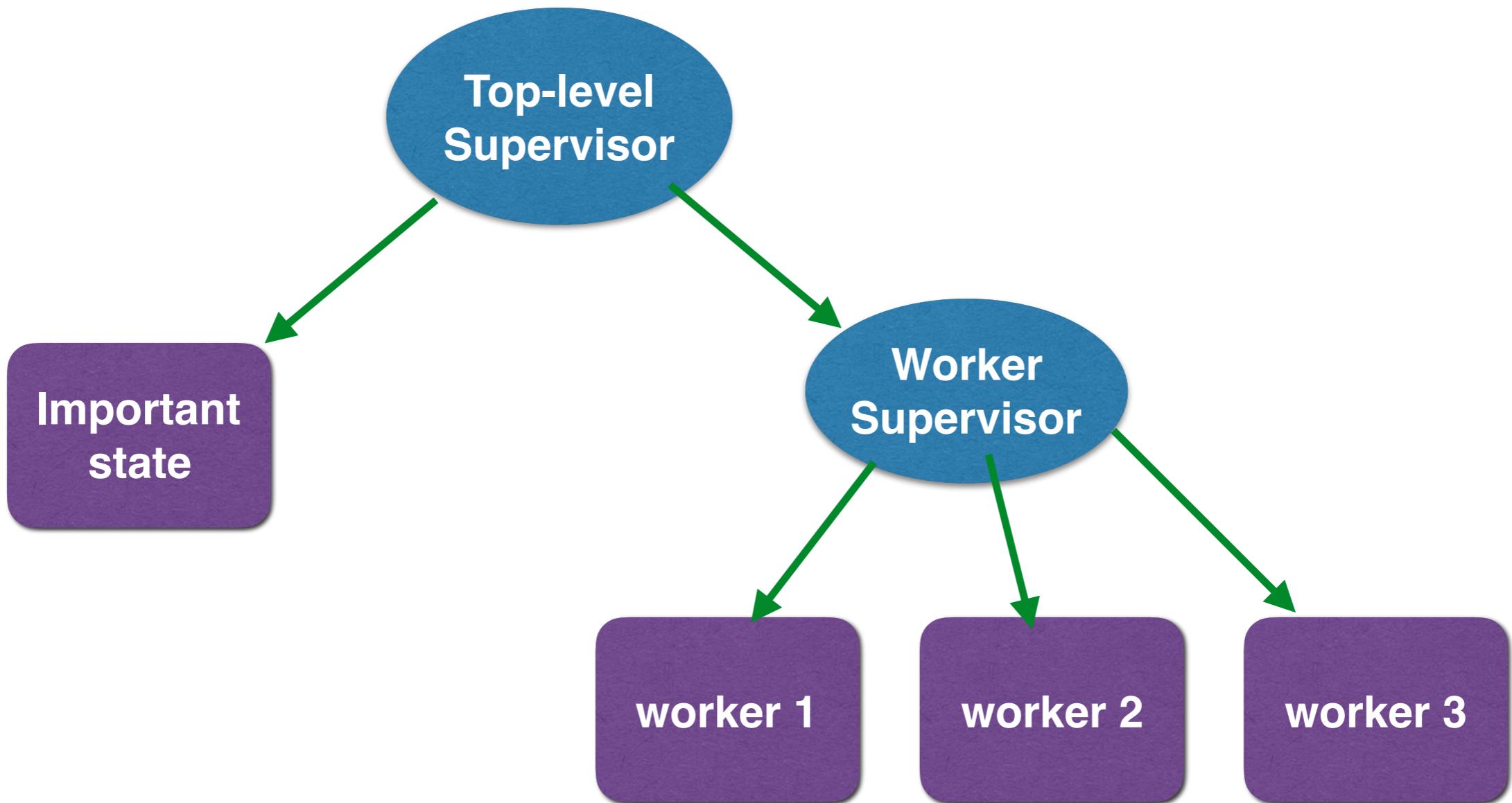
Supervision

- Supervisors do not form part of the logic of the application.
- Supervision structure is separate from the application processes
- You ask a supervisor to start processes, but you then talk directly to those processes

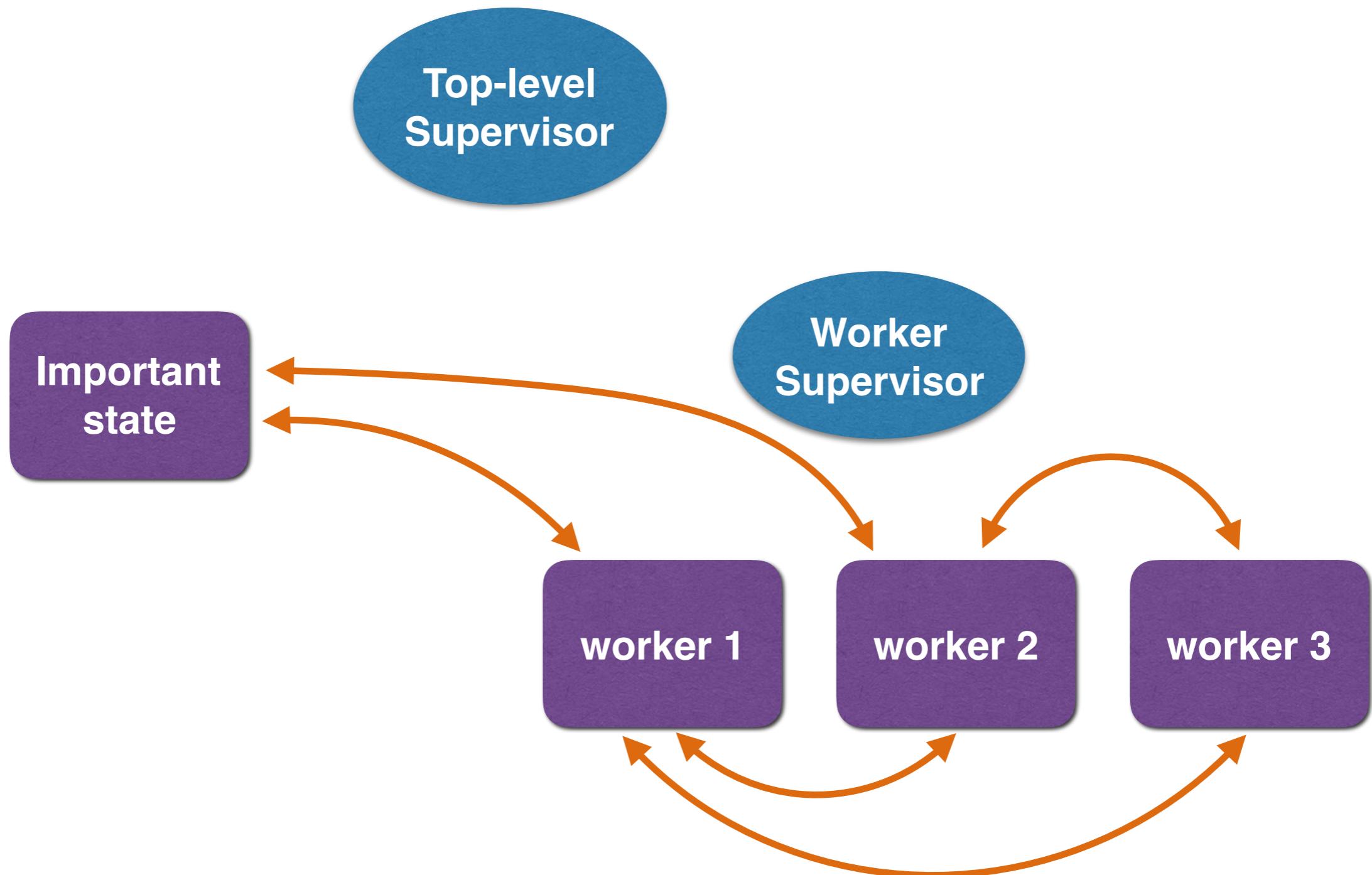
Supervision

- How to structure
 - don't over-supervise. You can start with `start_link` and supervise if you see a need
 - remember you can supervise supervisors to build trees
 - store valuable state nearer the top of the tree, and stuff than can break at the bottom

Supervision Structure



Interaction Structure



Worker Pools

- The list of children you pass to a supervisor is static.
- The `:simple_one_for_one` strategy lets you overcome this
- Limitation—all workers have the same specification

```
defmodule TwitterFeed.FetcherSupervisor do
  @me :fetcher_supervisor

  def start_link do
    import Supervisor.Spec

    children = [
      worker(TwitterFeed.FetcherServer, [], restart: :transient)
    ]

    Supervisor.start_link(children, name: @me, strategy: :simple_one_for_one)
  end

  def new_fetcher(params) do
    Supervisor.start_child(@me, params)
  end
end
```

Applications

Applications

- Are not what ROW calls apps
- distributable, free-standing chunk of code and metadata
- can be started/stopped independently
- or in the buzzword-du-jour...

SERVICES!

Turning Project into App

- Provide an entry point
 - `start(_type, _args)`
 - normally in top-level module
- Update application section of mix.exs to point to it.

```
def application do
  [
    mod: { Hangman, [] },
  ]
end
```

App Startup

- When you build project, mix generates a file called «appname».app. This tells BEAM how to run your code
- BEAM loads the module you specified with mod: and runs its start function.
- Typically this function kicks off the top-level supervisor

```
def start(_type, _args) do
  import Supervisor.Spec
  children = [
    supervisor(CalcHistorySupervisor, []),
    worker(Summer, [])
    worker(TakeAwayer, [])
    worker(Divider, [])
    worker(Multiplier, [])
  ]
  Supervisor.start_link(children, strategy: :one_for_one)
end
```

name of sub supervisor module

```
def application do
  [
    mod: { Hangman, [] },
    extra_applications: [
      :logger,
    ],
  ]
end
```



Most third-party applications are started automatically, because mix sees them as dependencies

Some applications, such as Logger, are supplied with Elixir, so you need to tell mix explicitly if you want them started.

Lab

Create a supervised application

Phoenix



Phoenix Framework

- Written in Elixir
- OTP Application
- Scalable
- Reliable
- Modern



PHOENIX

IS

NOT

RAILS

Phoenix

- Set of components to let you interface back-end services to front-end clients
 - HTML → DB-backed model (Rails)
 - Single-page app → Pure services
 - Single-pageapp → Single-page app
 - ...



Install Fest Time!



Follow instructions at:

<http://www.phoenixframework.org/docs/installation>

tl;dr;

1. `mix archive.install https://github.com/phoenixframework/archives/raw/master/phoenix_new.ez`
2. Install node.js
3. DO NOT bother with Postgresql
4. Linux users install inotify

```
$ mix phoenix.new test --no-brunch --no-ecto
* creating test/config/config.exs
* creating test/config/dev.exs
...
* creating test/web/views/layout_view.ex
* creating test/web/views/page_view.ex
```

Fetch and install dependencies? [Yn] y

(takes a while...)

```
$ cd test
```

```
$ mix phoenix.server
```

(installs and builds dependencies)

[info] Running Test.Endpoint with Cowboy using
<http://localhost:4000>

A screenshot of a web browser window displaying the Phoenix Framework homepage. The browser's address bar shows "localhost:4000". The page features the Phoenix logo (a red stylized bird) and the text "Phoenix Framework". A large central box contains the heading "Welcome to Phoenix!" and the subtext "A productive web framework that does not compromise speed and maintainability." Below this, sections for "Resources" and "Help" are listed with links.

localhost:4000

Phoenix Framework

Get Started

Welcome to Phoenix!

A productive web framework that does not compromise speed and maintainability.

Resources

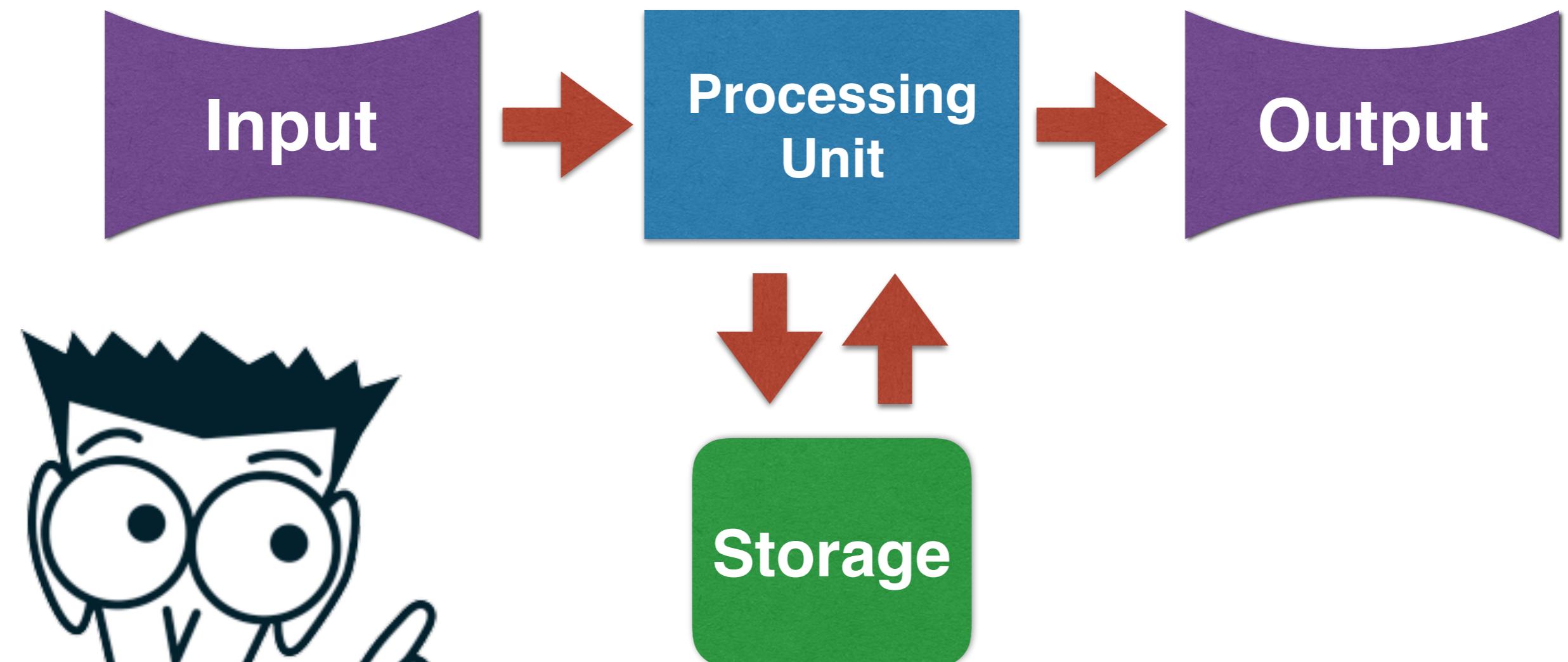
- [Guides](#)
- [Docs](#)
- [Source](#)

Help

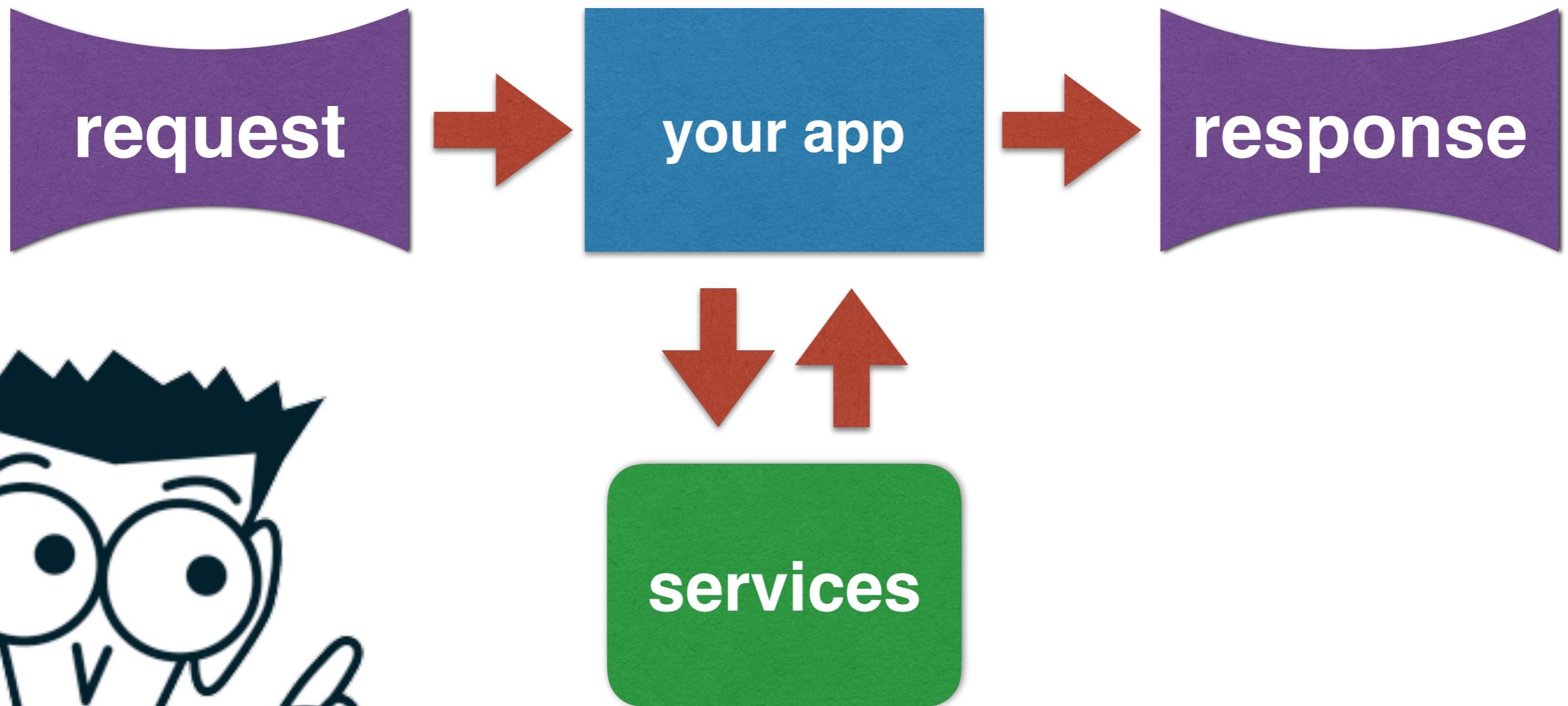
- [Mailing list](#)
- [#elixir-lang on freenode IRC](#)
- [@elixirphoenix](#)

What Phoenix Does

How Computers Work



How Phoenix Works



How it Does It

PLUG





AUSTRALIA





Forbes ranks Happiness



#2



#4



#12



Plug

- Both a DSL and an implementation
- DSL builds chains of (nested) plugs that will be used to transform a particular request
- Implementation then shepherds a request through this chain until a result is produced

incoming
request

```
defmodule D.Endpoint do
  use Phoenix.Endpoint, otp_app: :d

  socket "/socket", D.UserSocket

  plug Plug.Static,
    at: "/", from: :d, gzip: false,
    only: ~w(css fonts images js favicon.ico robots.txt)

  plug Plug.RequestId
  plug Plug.Logger

  plug Plug.Parsers,
    parsers: [:urlencoded, :multipart, :json],
    pass: ["*/*"],
    json_decoder: Poison

  plug Plug.MethodOverride
  plug Plug.Head

  plug Plug.Session,
    store: :cookie,
    key: "_d_key",
    signing_salt: "ZQ3hAfq1"

  plug D.Router
end
```

lib/my_app/endpoint.ex

web/router.ex

```
defmodule MyApp.Router do
  use MyApp.Web, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", MyApp do
    pipe_through :browser # Use the default browser stack

    get "/", PageController, :index
  end
end
```

```
defmodule MyApp.Router do
# . . .

scope "/", MyApp do
  pipe_through :browser # Use the default browser stack

  get "/", PageController, :index
end
end
```

web/controllers/page_controller.ex

Yes, this is
effectively
a plug, too!

```
defmodule MyApp.PageController do
  use MyApp.Web, :controller

  def index(conn, _params) do
    render conn, "index.html"
  end
end
```

```
<div class="jumbotron">
  <h2><%= gettext "Welcome to %{name}", name: "Phoenix!" %></h2>
  <p class="lead">A productive web framework that<br />does not compromise speed and maintainability.</p>
</div>

<div class="row marketing">
  <div class="col-lg-6">
    <h4>Resources</h4>
    <ul>
      <li>
        <a href="http://phoenixframework.org/docs/overview">Guides</a>
      </li>
      <li>
        <a href="https://hexdocs.pm/phoenix">Docs</a>
      </li>
      <li>
        <a href="https://github.com/phoenixframework/phoenix">Source</a>
      </li>
    </ul>
  </div>

  <div class="col-lg-6">
    <h4>Help</h4>
    <ul>
      <li>
        <a href="http://groups.google.com/group/phoenix-talk">Mailing list</a>
      </li>
      <li>
        <a href="http://webchat.freenode.net/?channels=elixir-lang">#elixir-lang on freenode IRC</a>
      </li>
      <li>
        <a href="https://twitter.com/elixirphoenix">@elixirphoenix</a>
      </li>
    </ul>
  </div>
</div>
```



Channels

- Where the future lies
- Persistent end-to-end connections (typically between client and Phoenix)
- Transport independent, but currently uses
 - WebSockets
 - long polling

Channel Architecture

- The channel is the transport connection
- Each channel multiplexes one or more logical sessions, called *topics*.
- Each topic is handled by a channel controller

Channel Flow

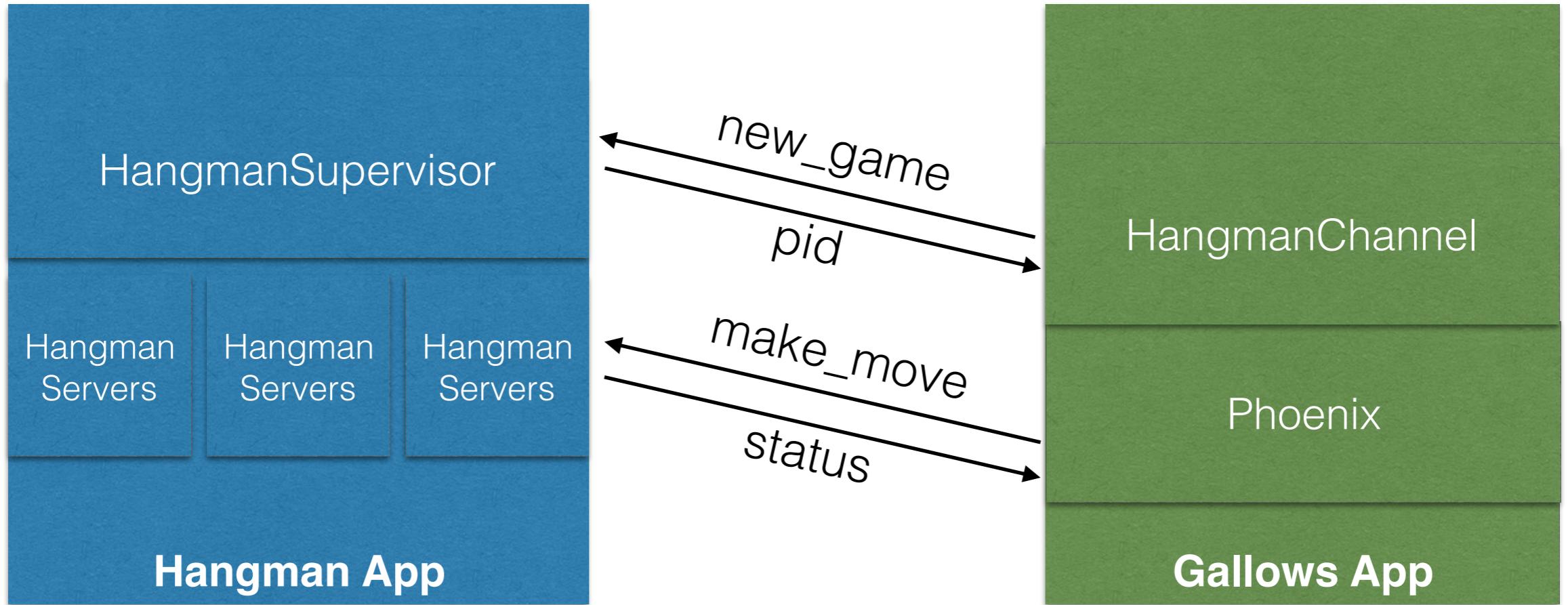
- Client joins a topic on a channel
- Channel controller authorizes
- Either end can send messages
 - channel controller receives client messages via handle_in callback
- Channel controller maintains state in a **socket** structure (passed to all handlers)

Hangman

- We've written a server application. We won't make any changes to it
- Now we'll write a front end using web-sockets talking to a view layer in the browser

Download

<https://github.com/....>



client Javascript

```
join_channel() {  
  let socket = new Socket("/socket", { logger: Client.my_logger })  
  socket.connect()  
  let channel = socket.channel("status")  
  channel.join()  
  return channel  
}
```

web/channels/user_socket.ex

```
defmodule Gallows.UserSocket do  
  use Phoenix.Socket  
  
  channel "hangman:*", Gallows.HangmanChannel
```

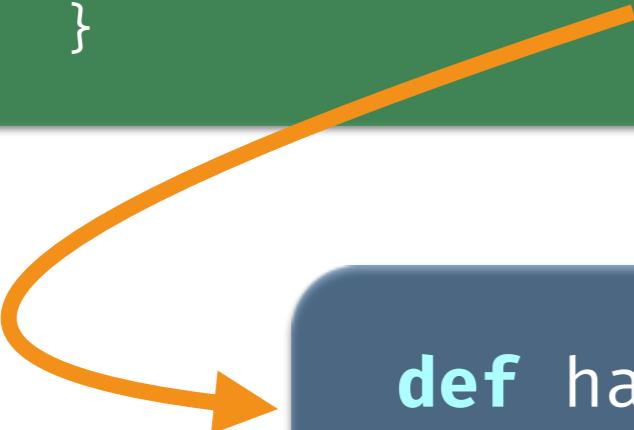
server web/channels/hangman_channel.ex

```
defmodule Gallows.HangmanChannel do  
  use Gallows.Web, :channel  
  alias Hangman.GameServer, as: Game  
  
  def join("hangman:game", _payload, socket) do  
    {:ok, assign(socket, :game, Hangman.GameSupervisor.new_game)}  
  end
```

client Javascript

```
constructor() {  
  this.setupDOM()  
  this.channel = this.join_channel(); ✓  
  this.setupEventHandlers(this.channel)  
  this.gallows = new Gallows()  
  this.channel.push("get_status", {})  
}
```

hangman_channel.ex



```
def handle_in("get_status", _, socket) do  
  Game.get_status(socket.assigns.game)  
  ▷ reply_with_status(socket)  
end  
  
defp reply_with_status(status, socket) do  
  push socket, "status", status  
  { :noreply, socket }  
end
```

```
setupEventHandlers(channel) {  
    channel.on("status", msg => this.update_status(msg))  
    this.again.on("click", (ev) => this.play_again(ev))  
    this.letters.on("click", (ev) => this.handle_click_on_letter(ev))  
    $(document).on("keyup", (ev) => this.handle_keypress(ev))  
}
```

```
update_status(msg) {  
    this.guesses_left.text(" " + msg.turns_left)  
    this.word_so_far.text(msg.letters.join(" "))  
  
    $(`#letter-${msg.last_guess}`).addClass(msg.guess_state)  
  
this.gallows.display_for(msg.turns_left)  
  
if (msg.game_state != "in_progress") { // get here if we have either won or lost  
    this.letters.addClass("done")  
  
if (msg.game_state == "won") {  
    this.gallows.display_win()  
}  
else {  
    this.gallows.display_loss()  
}  
}  
}
```



client Javascript

```
setupEventHandlers(channel) {  
    channel.on("status", msg => this.update_status(msg))  
    this.again.on("click", (ev) => this.play_again(ev))  
    this.letters.on("click", (ev) => this.handle_click_on_letter(ev))  
    $(document).on("keyup", (ev) => this.handle_keypress(ev))  
}
```

```
handle_click_on_letter(event) { ←  
    let letter = event.target  
    $(letter).addClass("guessed")  
    event.preventDefault();  
    this.channel.push("guess", { letter: letter.text })  
}
```

```
this.channel.push("guess", { letter: letter.text })
```

hangman_channel.ex

```
def handle_in("guess", %{ "letter" => letter }, socket) do
  Game.make_move(socket.assigns.game, letter)
  ▷ reply_with_status(socket)
end
```

```
update_status(msg) {
  ...
}
```