Programming Elixir

My Opinions…

Dave Thomas
@pragdave
dave@pragdave.me

https://github.com/elixir-lab/slides

Programming

is **transforming** data

Design

make programs

easy to change

```
def get_[illegible]note_numbers(refs, footnotes, options) do
    Enum.reduce([illegible]efs, [], fn(ref, list) ->
      case Enum.find([illegible]otnotes, &(&1.id == ref))[illegible]
        note = %Block.FnDef[illegible] -> number = [illegible]gth(list) + options.footnote_offset
                                          [illegible] %Block.FnDef{ note | number: number }
                                          [illegible] list ]
                                       -> list # [illegible]line 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

Looks are Everything

But First…

Tools

`elixir` compile and run

`mix` build tool + …

`iex` interactive elixir shell

Files

`wilma.ex` source: intended to be part of project

`fred.exs` script: run, don't save

wilma.ex

```
IO.puts “Chaaaarge it!!”
```

```
$ elixir wilma.ex
Chaaaarge it!!
$
```

Statements: line oriented. Continue across lines when obvious:

```
IO.puts a + 3


IO.puts a +
        3


IO.puts(
   a
   +
   3
   )
```

Variable & function names:

```
abc   ab_c99    hungry?    yes!
```

Module names must start upper case. Namespace using "."

```
Hangman.Logic.make_move()
```

Blocks: (not really, but…)

```
do
   …              or       do: …
end
```

Comments:

```
name = String.upcase(id)  # normalize all names
```

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
```

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

```elixir
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
```

Lists

```
[ a, b, c, d ] = [ 1, 2, 3, 2 ]

a → 1

b → 2

. . .
```

Quiz

Basic Pattern Matching

`a = 123`	a = ?
`b = 27` `a = 3 + b`	a = ?
`[ a, b ] = [ 3, 4, 5 ]`	a = ?, b = ?
`[a, b] = [ 3, 5 ]`	a = ?, b = ?
`a + 1 = 5`	a = ?
`[ a, a ] = [ 3, 4 ]`	a = ?
`[ x, [ a, b ] ] = [ 1, [ 2, 3] ]`	a = ?, b = ?, x = ?
`[ a, a-1 ] = [ 3, 2 ]`	a = ?
`[ b, a, b, c ] = [ 0, 1, 2-2, 3 ]`	a = ?,b = ?, c= ?

Arbitrary Lists

A list is:

- empty, or
- a value followed by a list

Arbitrary Lists

A list is:

- empty, or
- a value followed by a list

- [ ]
- [value | a_list]

Arbitrary Lists

A list is:

- empty, or
- a value followed by a list

- [ ]
- [head | tail]

Arbitrary Lists

```
[ h | t ] = [ 1, 2, 3, 2 ]

h → 1

t → [ 2, 3, 4 ]
```

```
[ h1, h2 | t ] = [ 1, 2, 3, 2 ]

h1 → 1

h2 → 2

t → [ 3, 4 ]
```

Build Lists

```
rest = [ 2, 3, 4 ]

[ 1 | rest ]   # ⇒ [ 1, 2, 3, 4 ]
```

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
```

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

```elixir
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
```

```elixir
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
```

```elixir
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)
```

```elixir
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
```

Pipelines

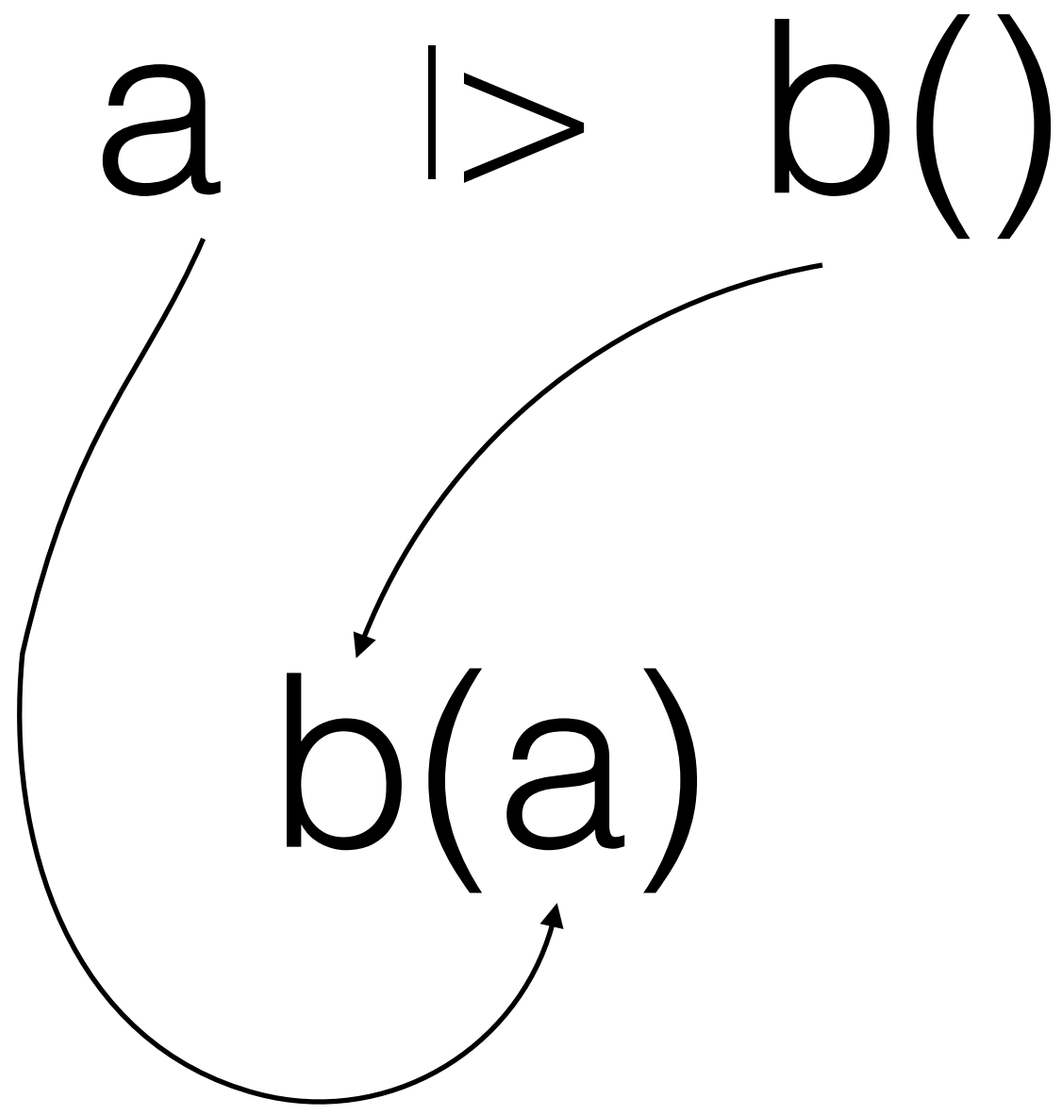
Code is Transformations

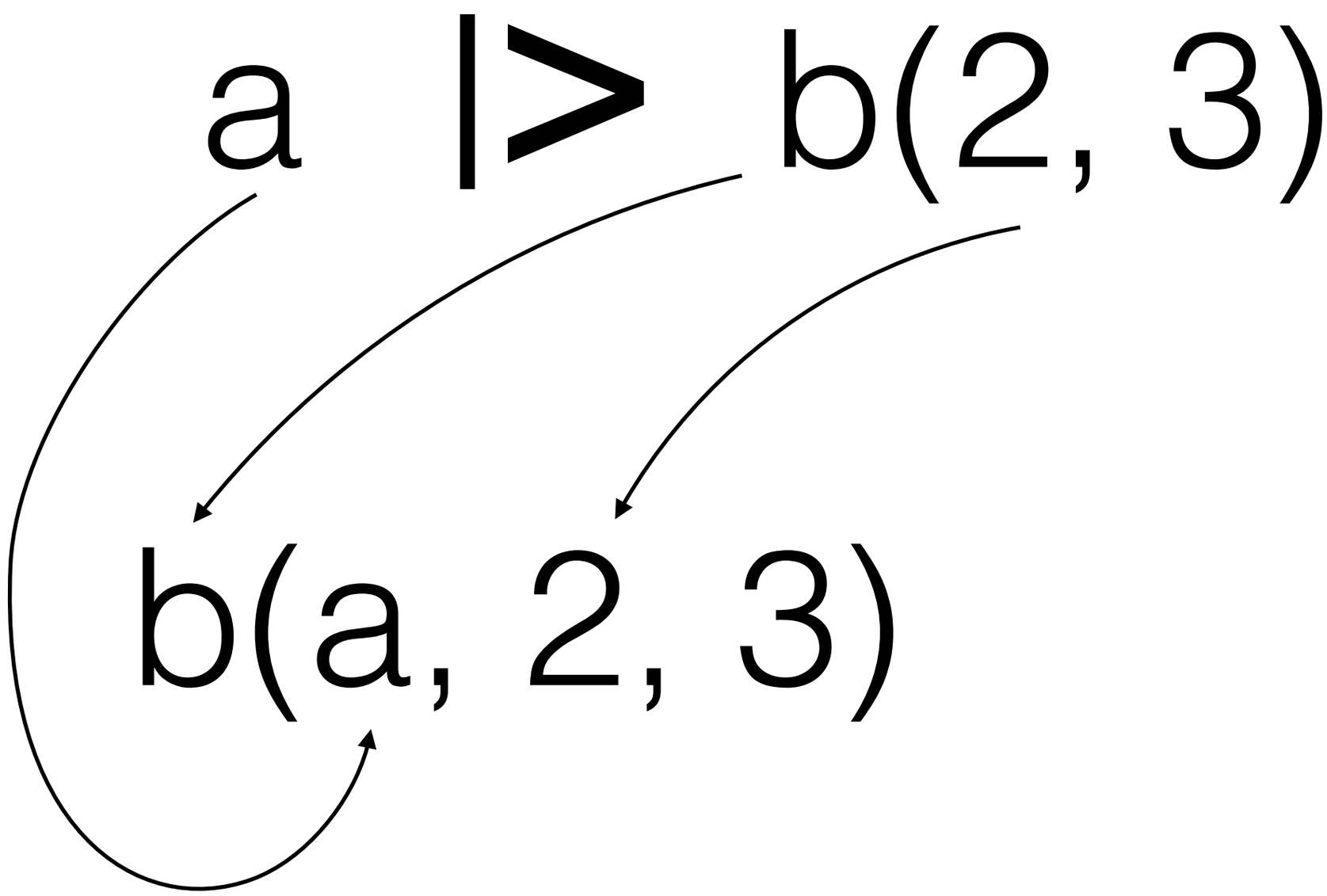
```
user   = find_user(name)
orders = get_orders(user)
total  = order_total(orders)
IO.puts total
```

```
IO.puts(order_total(get_orders(find_user(name))))
```

```
name |> find_user() |> get_orders() |> order_total() |> IO.puts()
```

```
name
|> find_user()
|> get_orders()
|> order_total()
|> IO.puts()
```


a |> b()
b(a)


a |> b(2, 3)
b(a, 2, 3)

The Ideal

A function is:

- a single pipeline
- no local variables (except parameters)
- no conditional code

Lab

Time to Tidy

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

Projects

(think components)

What is a Project?

- a set of conventions
    - directory structure
    - file naming
    - file contents
- a set of tools that assume the above

Why use Projects?

- Interoperability
- Tool support
- Communicate with other people
- Why reinvent the wheel?

Names

- Erlang calls projects "applications." They aren't.
- Think of a project as a component. One of more components work together to make the functionality of an application.

'When I use a word,'
Humpty Dumpty said, in rather
a scornful tone, 'it means just what I choose
it to mean — neither more nor less.'

"dictionary" project

```
dictionary/
├── README.md
├── config
│   └── config.exs
├── lib
│   └── dictionary.ex
├── mix.exs
└── test
    ├── dictionary_test.exs
    └── test_helper.exs
```

```
19·06·38> mix new dictionary
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/dictionary.ex
* creating test
* creating test/test_helper.exs
* creating test/dictionary_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd dictionary
    mix test

Run "mix help" for more commands.
```

Good Start To Any Project…

```
> mix new dictionary
* creating README.md
* creating .gitignore
. . .

> cd dictionary

> git init
Initialized empty Git repository in . . .

> git add .

> git commit -m 'some inspiring message'
[master (root-commit) 7f6074b] some inspiring message
 7 files changed, 114 insertions(+)

> mix test
Compiling 1 file (.ex)
Generated dictionary app
.
Finished in 0.02 seconds
1 test, 0 failures
```

"dictionary" project

```
dictionary/
├── README.md
├── config
│   └── config.exs
├── lib
│   └── dictionary.ex
├── mix.exs
└── test
    ├── dictionary_test.exs
    └── test_helper.exs
```

Your project's description.
Github displays it automatically.
Markdown format

"dictionary" project

```
dictionary/
├── README.md
├── config
│   └── config.exs
├── lib
│   └── dictionary.ex
├── mix.exs
└── test
    ├── dictionary_test.exs
    └── test_helper.exs
```

Compile-time configuration. Also used to differentiate development, test, and production modes

"dictionary" project

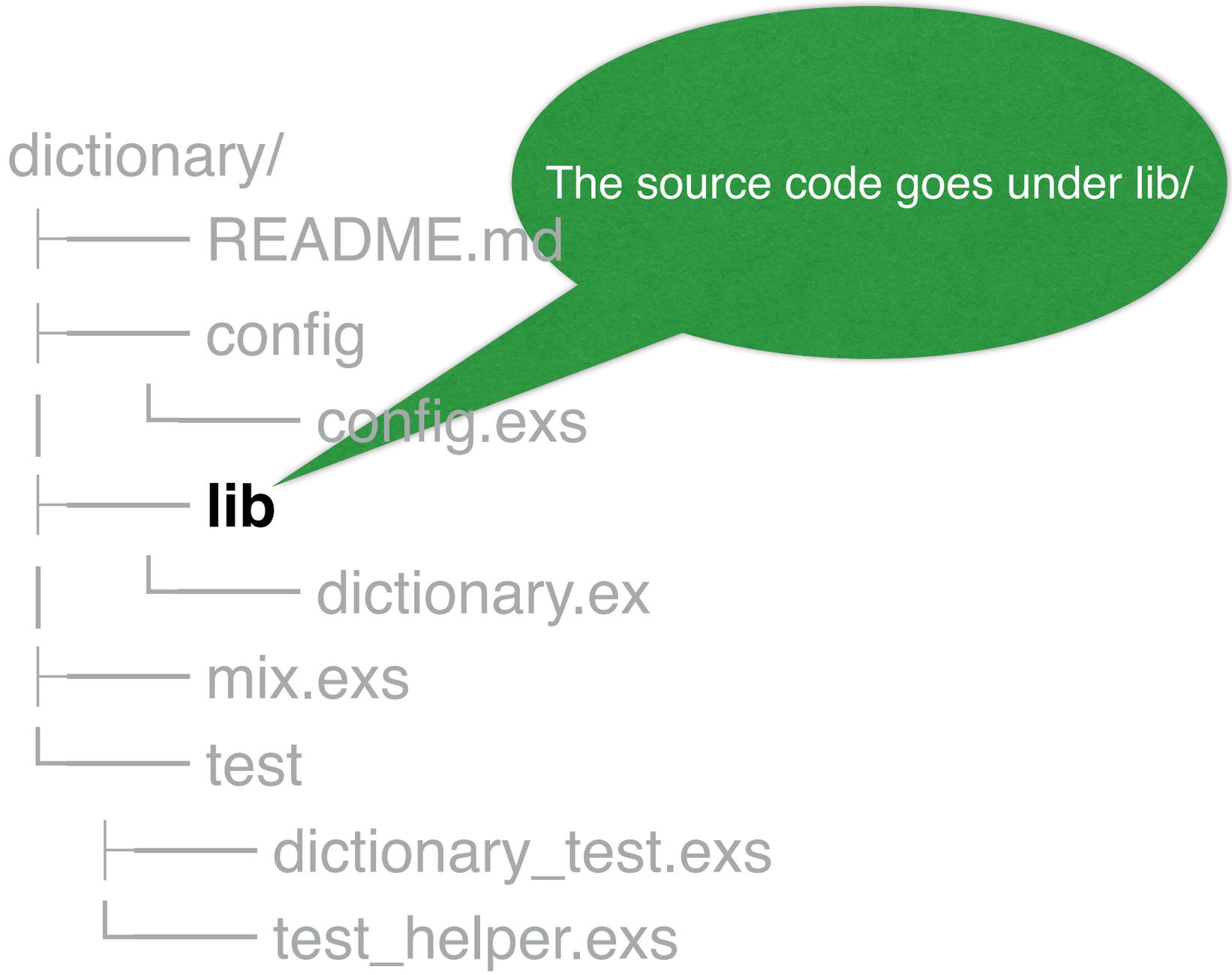
```
dictionary/
├── README.md
├── config
│   └── config.exs
├── lib
│   └── dictionary.ex
├── mix.exs
└── test
    ├── dictionary_test.exs
    └── test_helper.exs
```

Your project's metadata (version, dependencies, included applications, etc)

"dictionary" project

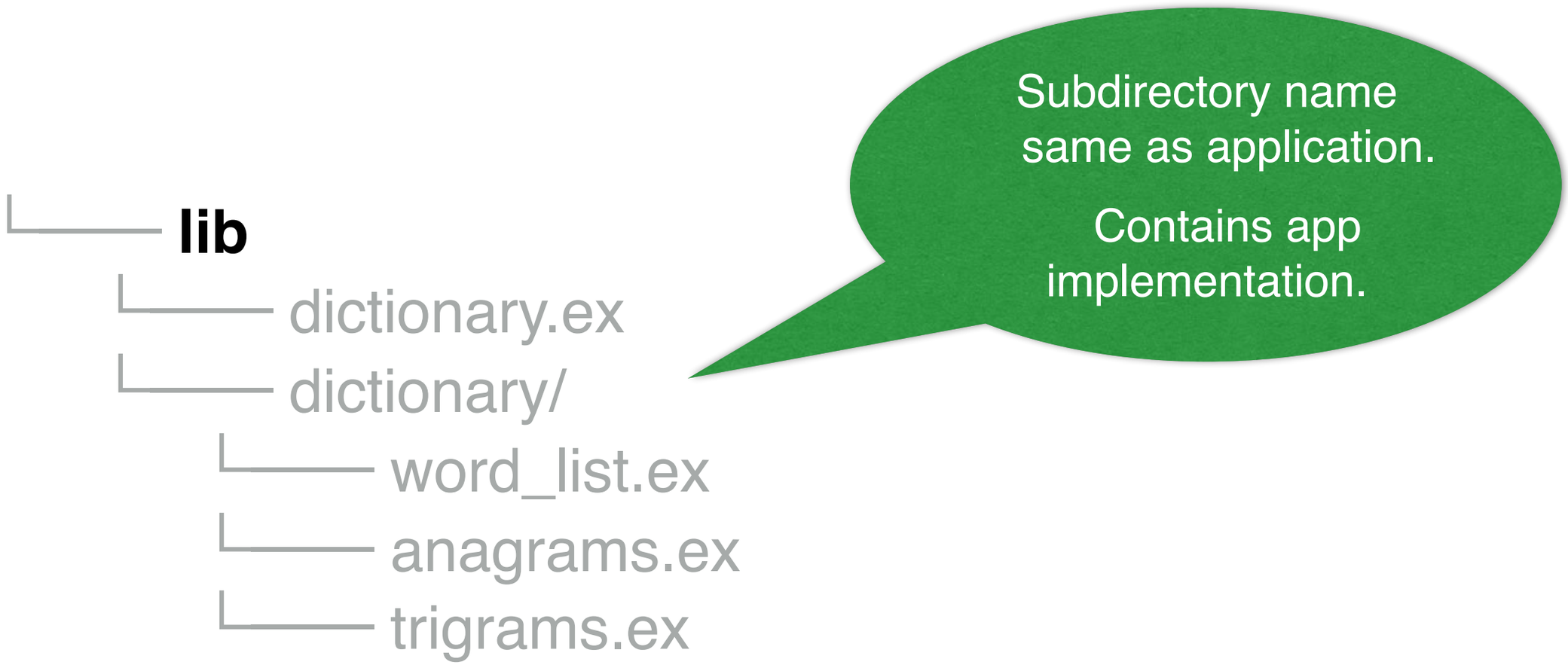
```
dictionary/
├── README.md
├── config
│   └── config.exs
├── lib
│   └── dictionary.ex
├── mix.exs
└── test
    ├── dictionary_test.exs
    └── test_helper.exs
```

"dictionary" project



```
└── lib
    ├── dictionary.ex
    ├── dictionary/
    │   ├── message.ex
    │   ├── photo.ex
    │   └── tweet.ex
```

```
defmodule Dictionary do
  # top level code
end
```


lib
dictionary.ex
dictionary/
word_list.ex
anagrams.ex
trigrams.ex
Subdirectory name same as application.
Contains app implementation.

```
└── lib
    ├── dictionary.ex
    ├── dictionary/
    │   ├── message.ex
    │   ├── photo.ex
    │   └── tweet.ex
```

```
defmodule Dictionary.Message do
  # code to implement messaging
end
```


lib
dictionary.ex
dictionary/
message.ex
photo.ex
tweet.ex
defmodule Dictionary.Message do
code to implement photos
end

For an application called MyApp

File & directory names	Module names
:	
└── lib	
├── my_app.ex	MyApp
└── my_app/	
├── parse_input.ex	MyApp.ParseInput
├── sales_tax.ex	MyApp.SalesTax
└── shipping.ex	MyApp.Shipping

File & directory names	Module names
my_app.ex	MyApp

- File names all lower case
- Separate words with underscores

- Module names mixed case
- Each word starts with an uppercase letter

File & directory names	Module names
my_app.ex	MyApp
my_app/	
├── parse_input.ex	MyApp.ParseInput
├── sales_tax.ex	MyApp.SalesTax
└── shipping.ex	MyApp.Shipping

When modules are in subdirectories, the subdirectory names are prepended to the module name, using the same conventions

mix.exs: Project Manifest

```elixir
defmodule Dictionary.Mixfile do
  use Mix.Project

  def project do
    [app:     :dictionary,
     version: "0.1.0",
     elixir:  "~> 1.3",
     build_embedded:  Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps:    deps()]
  end

  def application do
    [applications: [:logger]]
  end

  defp deps do
    []
  end
end
```

```elixir
defmodule Dictionary.Mixfile do
  use Mix.Project

  def project do
    [app:     :dictionary,
     version: "0.1.0",
     elixir:  "~> 1.3",
     build_embedded:  Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps:    deps()]
  end

  def application do
    [applications: [:logger]]
  end

  defp deps do
    []
  end
end
```

Metadata for building the project

```elixir
defmodule Dictionary.Mixfile do
  use Mix.Project

  def project do
    [app:     :dictionary,
     version: "0.1.0",
     elixir:  "~> 1.3",
     build_embedded:  Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps:    deps()]
  end

  def application do
    [applications: [:logger]]
  end

  defp deps do
    []
  end
end
```

Components to be run

```elixir
defmodule Dictionary.Mixfile do
  use Mix.Project

  def project do
    [app:     :dictionary,
     version: "0.1.0",
     elixir:  "~> 1.3",
     build_embedded:  Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps:    deps()]
  end

  def application do
    [applications: [:logger]]
  end

  defp deps do
    []
  end
end
```

External dependencies

Lab

Code along with Dave

as I create the dictionary app…

Lab

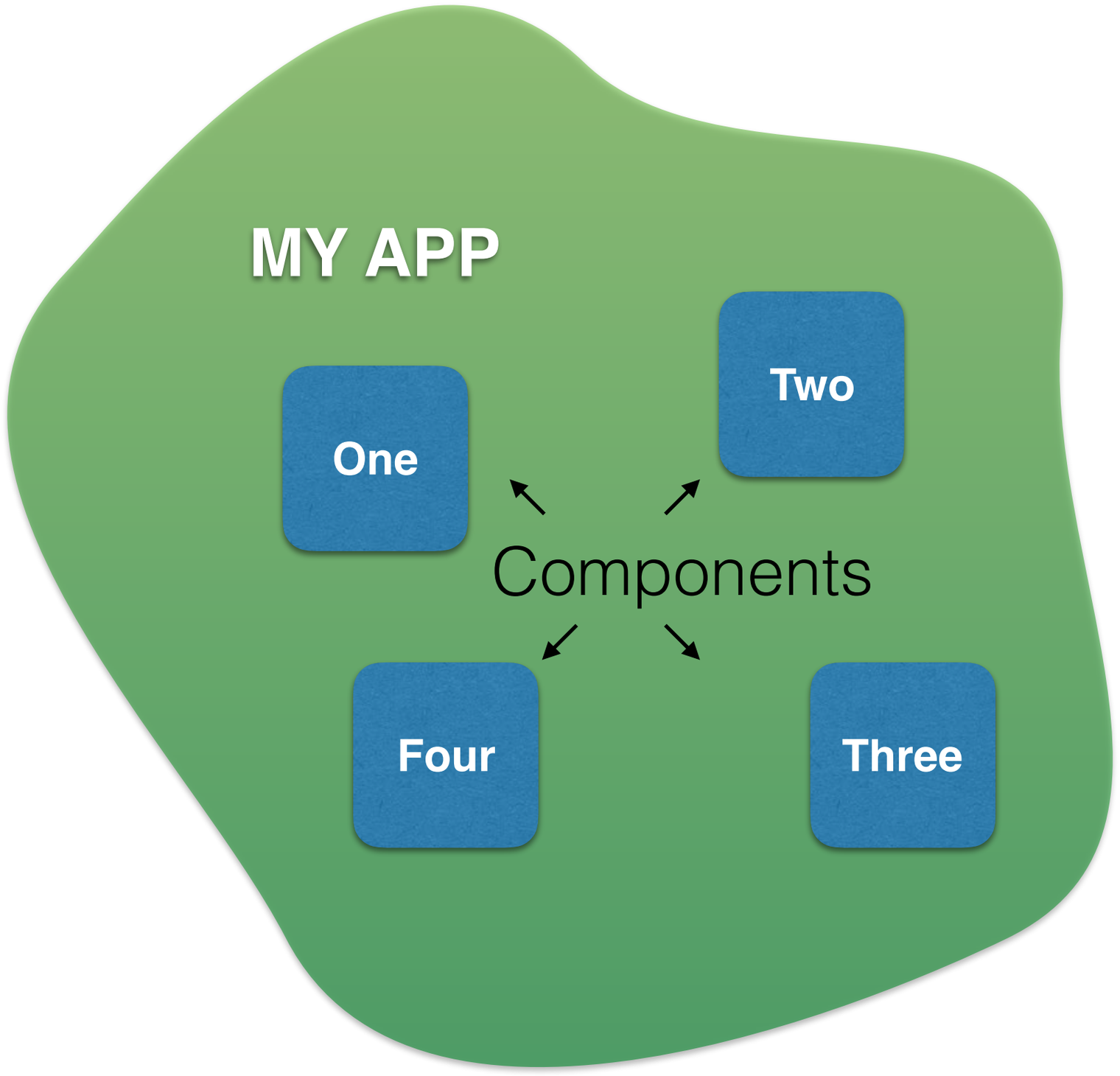
word file: https://goo.gl/6k2Azm

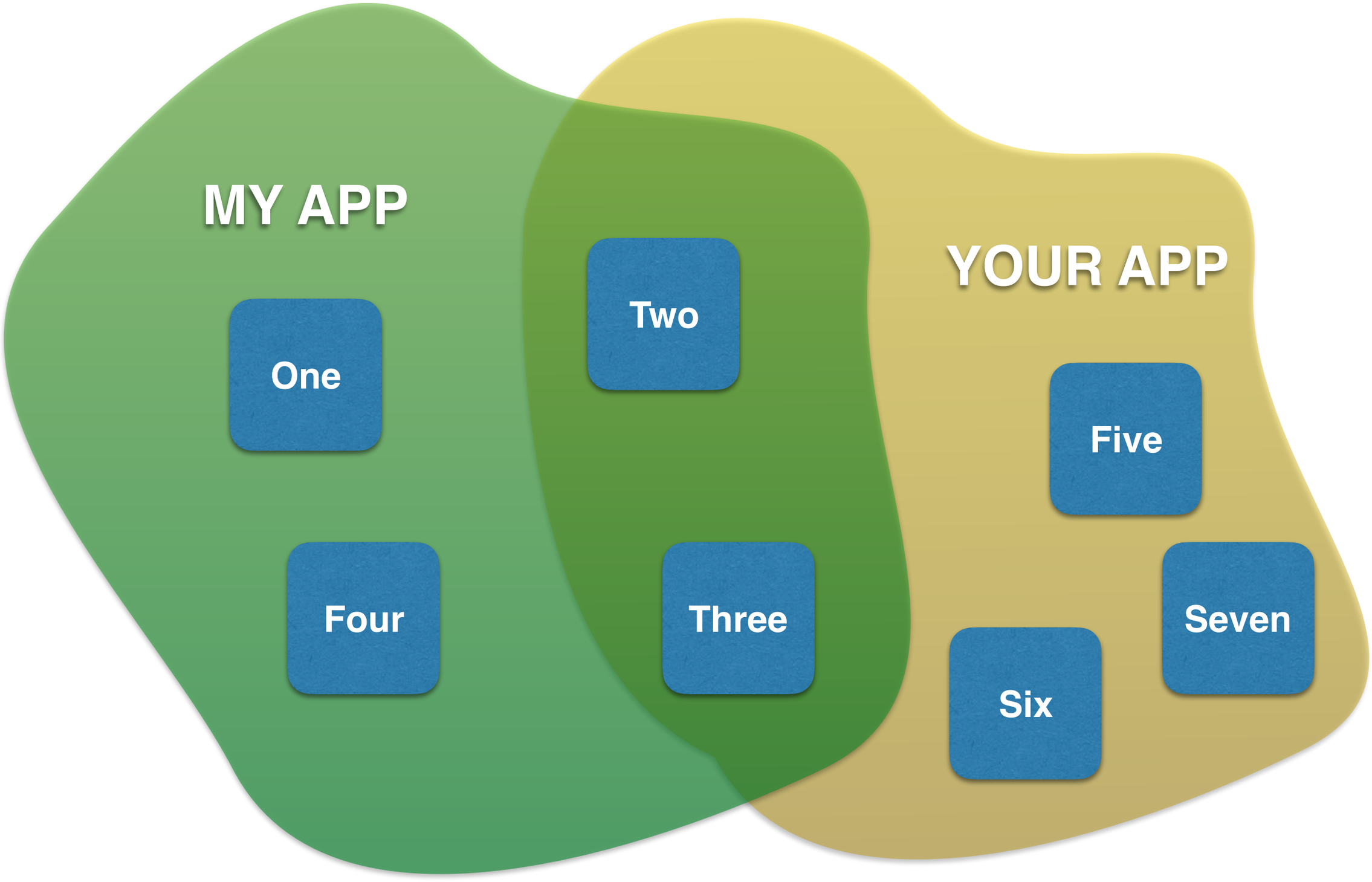
Dependencies

The end of the big ball of code

Elixir Projects are Components

Deliverable Applications are One or More Elixir Projects


MY APP
One
Two
Components
Four
Three


MY APP
YOUR APP
One
Two
Three
Four
Five
Six
Seven

```elixir
defmodule Dictionary.Mixfile do
  use Mix.Project

  def project do
    [app:     :dictionary,
     version: "0.1.0",
     elixir:  "~> 1.3",
     build_embedded:  Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps:    deps()]
  end

  def application do
    [applications: [:logger]]
  end

  defp deps do
    []
  end
end
```

Specify Dependencies in mix.exs

External dependencies

```elixir
defp deps do
  [
    { :jeeves, "⩾ 1.2.0" }
  ]
end
```

Fetch dependency "jeeves" from hex.pm

```
defp deps do
  [
    { :jeeves, "⩾ 1.2.0" },
    { :diet,   git: "https://github.com/pragdave/diet.git" }
  ]
end
```

Fetch dependency "diet"
from a Git URI

```
defp deps do
  [
    { :jeeves, "⩾ 1.2.0" },
    { :diet,   github: "pragdave/diet" }
  ]
end
```

Shortcut

```
defp deps do
  [
    { :jeeves, "⩾ 1.2.0" },
    { :diet,   github: "pragdave/diet" },
    { :dictionary, path: “../dictionary” },
  ]
end
```

Use Dependency

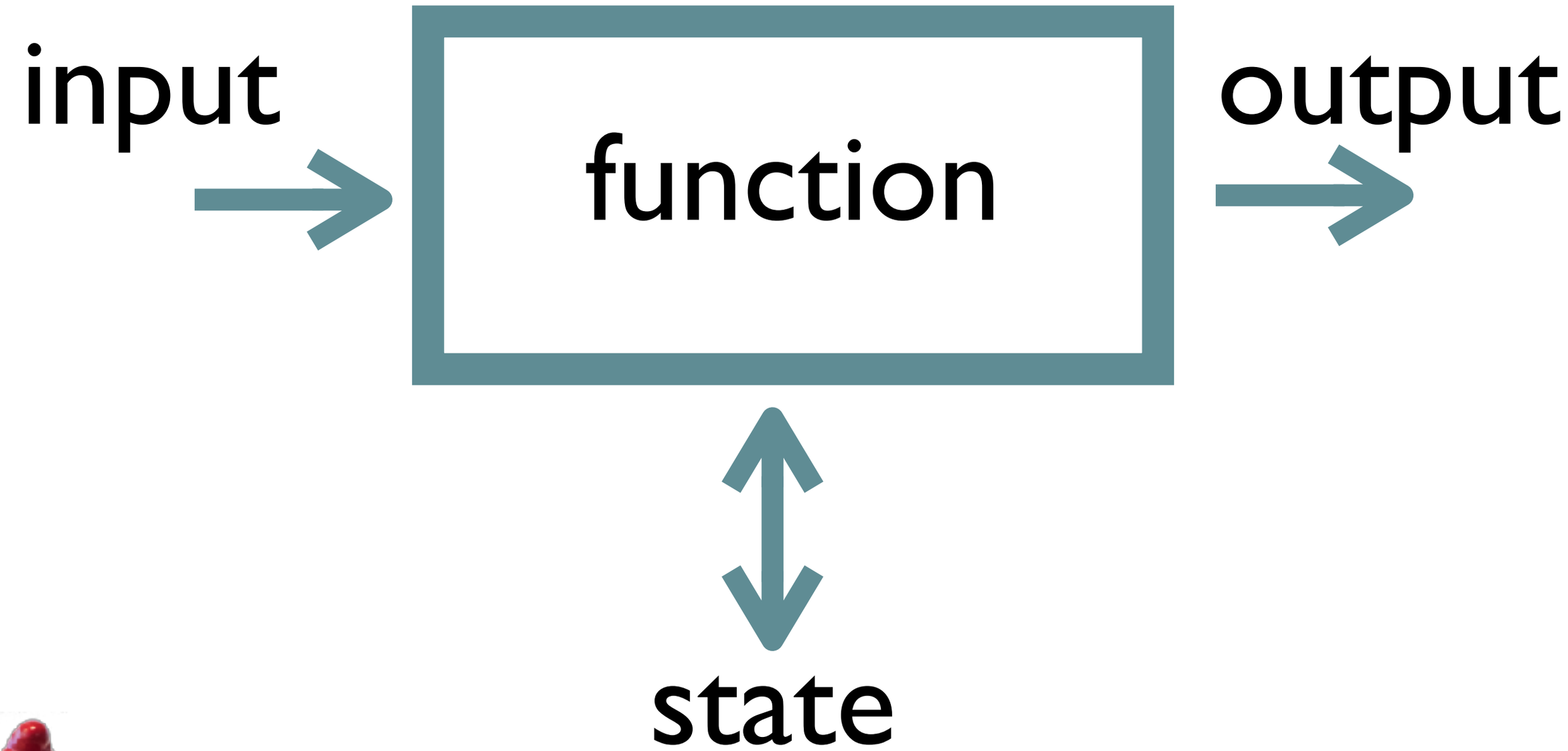
```
$ mix deps.get
```

- Then reference modules in dependency as if you'd written them

For example:

State in Practice

More that a Convenience


input
function
output
state

- 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

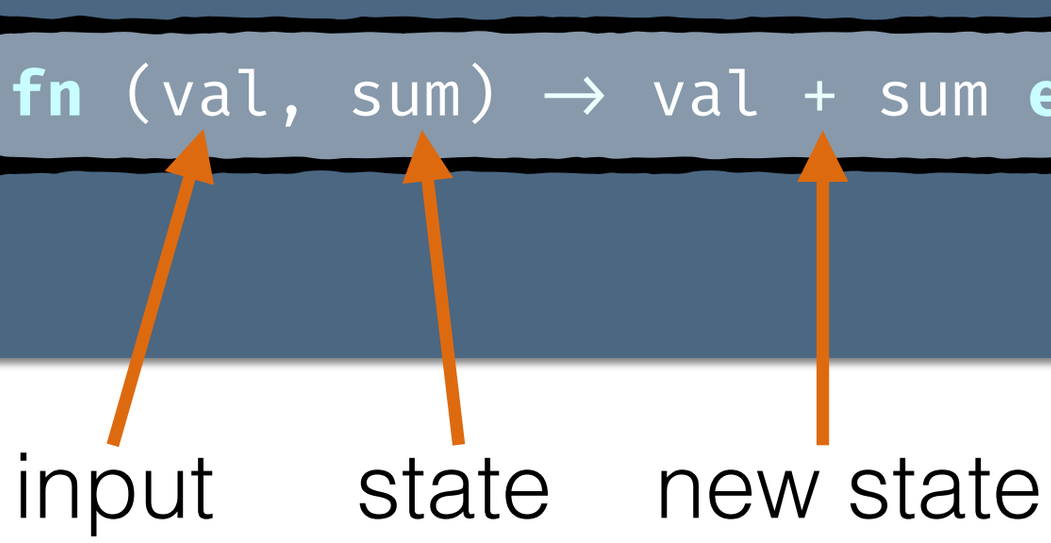
```
func1(state, inputs...) → new_state

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
```



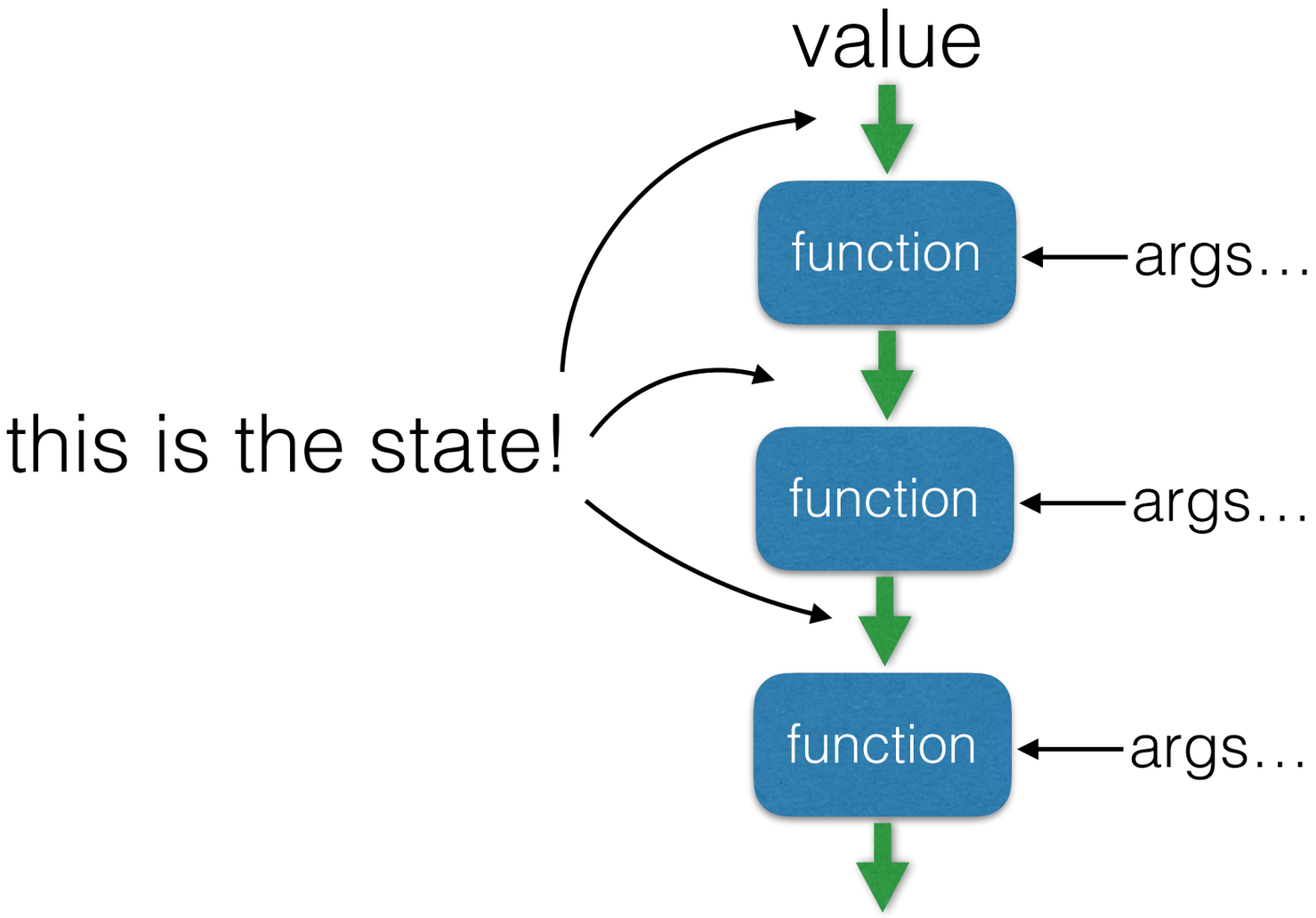
Reducers

- 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


value
function
args…
this is the state!
function
args…
function
args…

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+&2))
▷ IO.puts
```

So…

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

But...

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

The real world isn't a pipeline...

So make it one

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

```
def deduct(account, amount) do
  account
  |> Account.debit(amount)
  |> Account.update_credit_score()
  |> Audit.log(:deduct, amount)
end
```

But

```
def deduct(account, amount) do
  if Account.balance(account) < amount do
    {:insufficient_funds, account}
  else
    account
    |> Account.debit(amount)
    |> Account.update_credit_score()
    |> Audit.log(:deduct, amount)
  end
end
```

The real world has conditional logic

```elixir
def deduct(account, amount) do
  ok = Account.balance(account) < amount
  maybe_deduct(account, amount, ok)
end

def maybe_deduct(account, amount, true) do
  account
  |> Account.debit(amount)
  |> Account.update_credit_score()
  |> Audit.log(:deduct, amount)
end

def maybe_deduct(account, amount, false) do
  {:insufficient_funds, account}
end
```

```elixir
def deduct(account, amount) do
  if Account.balance(account) < amount do
    {:insufficient_funds, account}
  else
    account
    |> Account.debit(amount)
    |> Account.update_credit_score()
    |> Audit.log(:deduct, amount)
  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)

Hangman

API

```
Hangman.new_game()
```

→ `game` ← initial state

```
Hangman.make_move(game, letter)
```

→ `{ game', tally }`

updated state, result of move

```
Hangman.tally(game)
```

→ `tally`

Tally

```
%{
  game_state: :won | :lost | :good_guess | :bad_guess | :already_used,
  turns_left: 0..7,
  letters:    [ "w", "o", "_", "b", "a", "_" ],
  used:       [ "a", "b", "c", "h", "o", "w", "y"]
}
```

What is the State?

What is the State?

```
defmodule Hangman.Game do

  defstruct(
    turns_left: 7,
    game_state: :initializing,
    letters:    [],
    used:       MapSet.new(),
    last_guess: ""
  )

end
```

What is the State?

```
defmodule Hangman.Game do

  defstruct(
    turns_left: 7,
    game_state: :initializing,
    letters:    [],
    used:       MapSet.new(),
    last_guess: ""
  )

  def new_game(word) do
    %Hangman.Game{
      letters: word |> String.codepoints
    }
  end

  def new_game() do
    new_game(Dictionary.random_word)
  end

end
```

pattern match inside structure

```
def make_move(game = %{ game_state: state }, _guess)
when state in [:won, :lost] do
    game
    ▷ return_with_tally()
end
```

```
def make_move(game = %{ game_state: state }, _guess)
when state in [:won, :lost] do
    game
    ▷ return_with_tally()
end

 def make_move(game, guess) do
    game
    ▷ Map.put(:last_guess, guess)
    ▷ accept_move(MapSet.member?(game.used, guess))
    ▷ return_with_tally()
  end
```

Lab

Hangman using pattern matching

1. git clone https://github.com/elixir-lab/lab2
2. add code to lib/hangman/game.ex
3. `$ mix test`
4. Repeat 2 and 3 until tests pass
5. Try running using the text ui:

```
$ mix run -e Hangman.HumanPlayer.play
```

Processes

The rich-person's objects

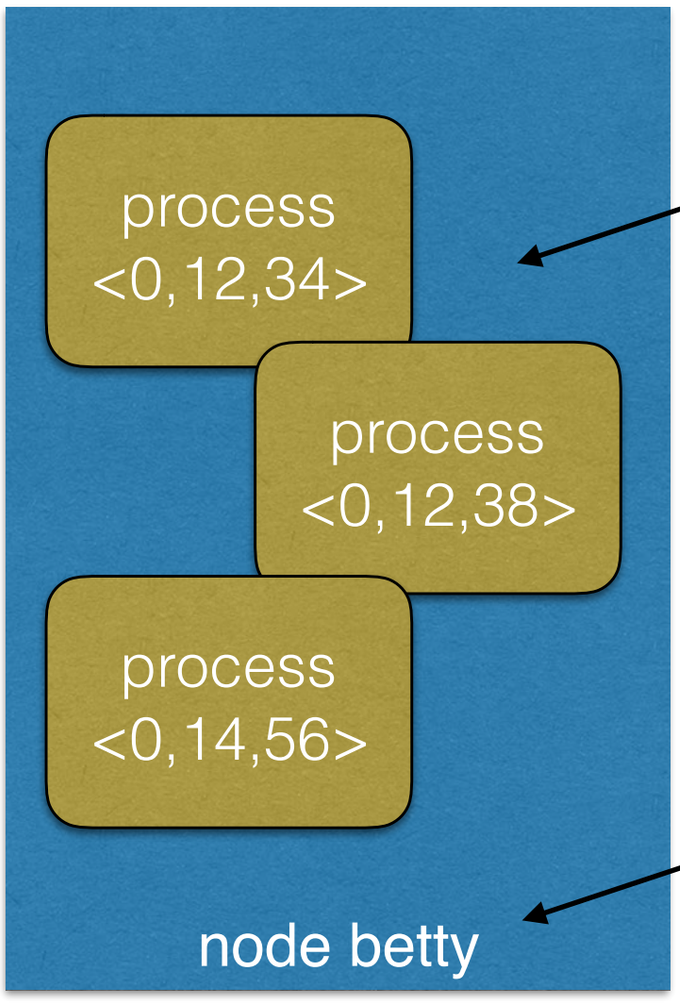
Why Processes?

- 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 run on the node

A node is an instance of the Erlang virtual machine

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

anonymous function

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

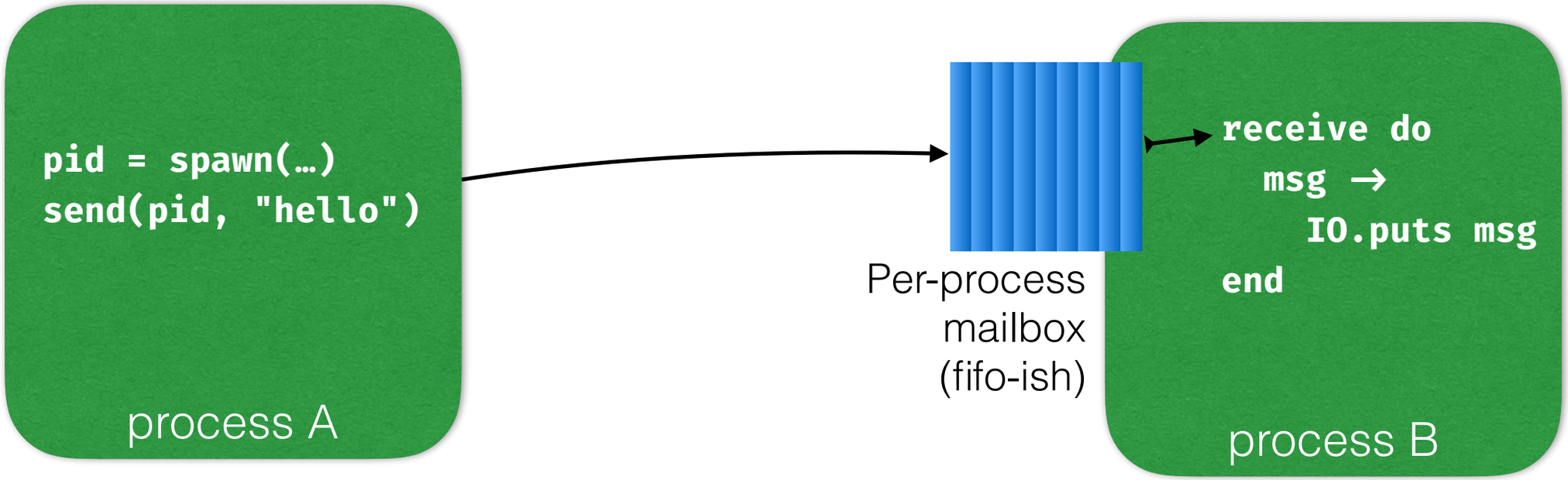
named function

```
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 ])
```

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
```

```elixir
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
```

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

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

```elixir
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

```elixir
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

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

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

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

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

```elixir
def listen(special) do
  receive do
   # do stuff…
  end
  listen(special)
end
```

special is passed between receipts of messages

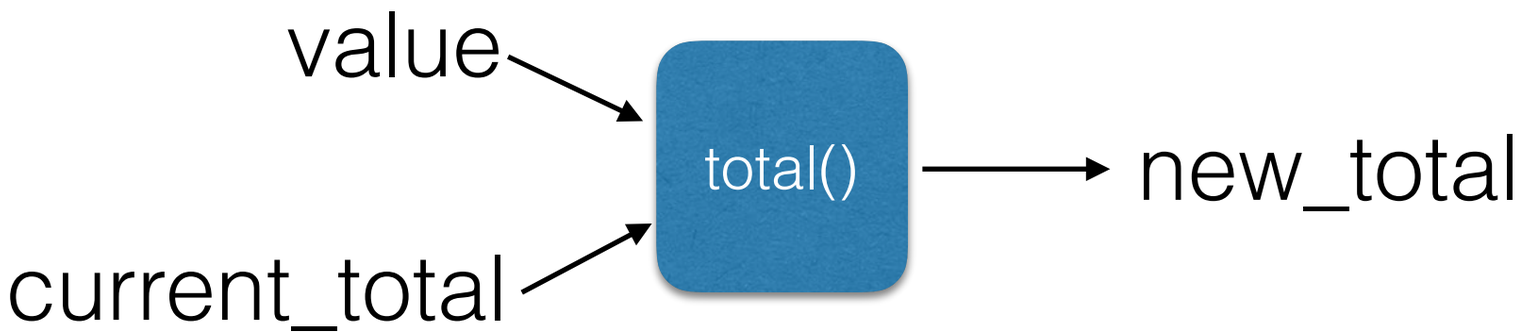
```elixir
def listen(special) do
  receive do
   # do stuff…
  end
  listen(special)
end
```

the STATE

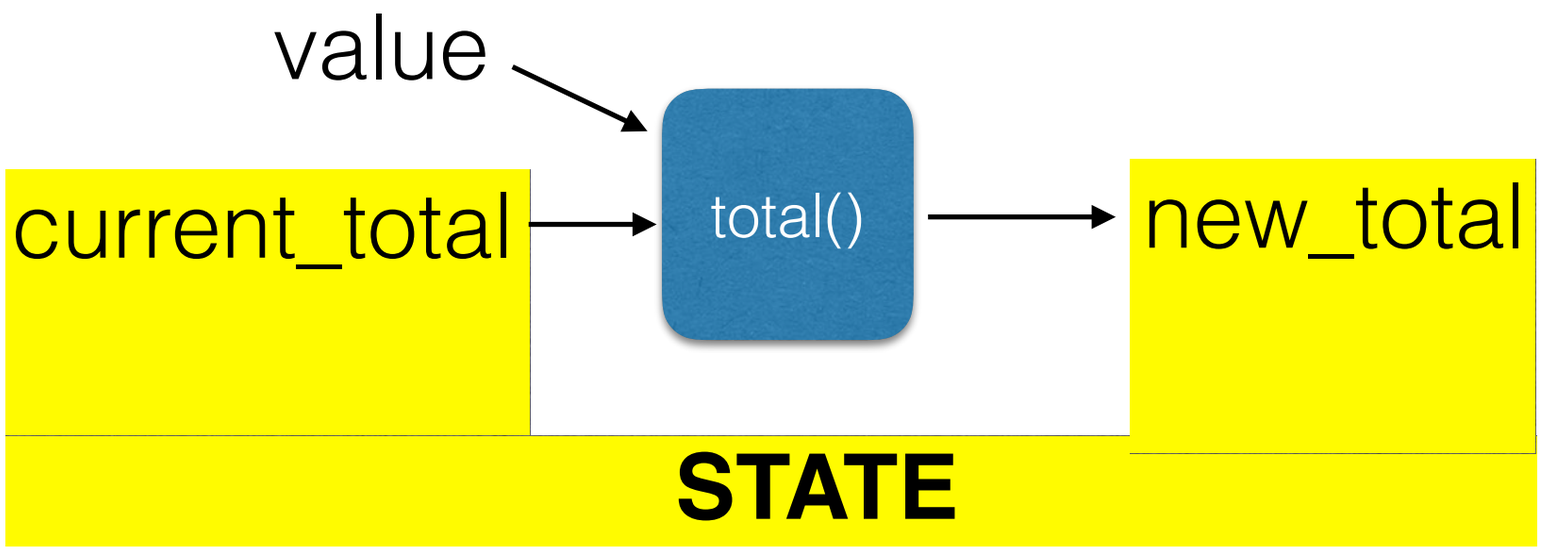
```elixir
def total(n \\ 0) do
  receive do
    {:add, value} when is_integer(value) ->
      IO.inspect [n, value]
      total(n+value)
    :done ->
      IO.puts "Total is #{n}"
  end
end
```

```
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
```

```elixir
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!



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
```

```elixir
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"
```

Lab

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

Playing with Processes

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

Just an atom…

```
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

Give Spawned Processes Names

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

iex> Process.register(pid, Echo)
true

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

Associate name with pid

Use the name in place of the pid

OTP GenSever 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

```elixir
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
```

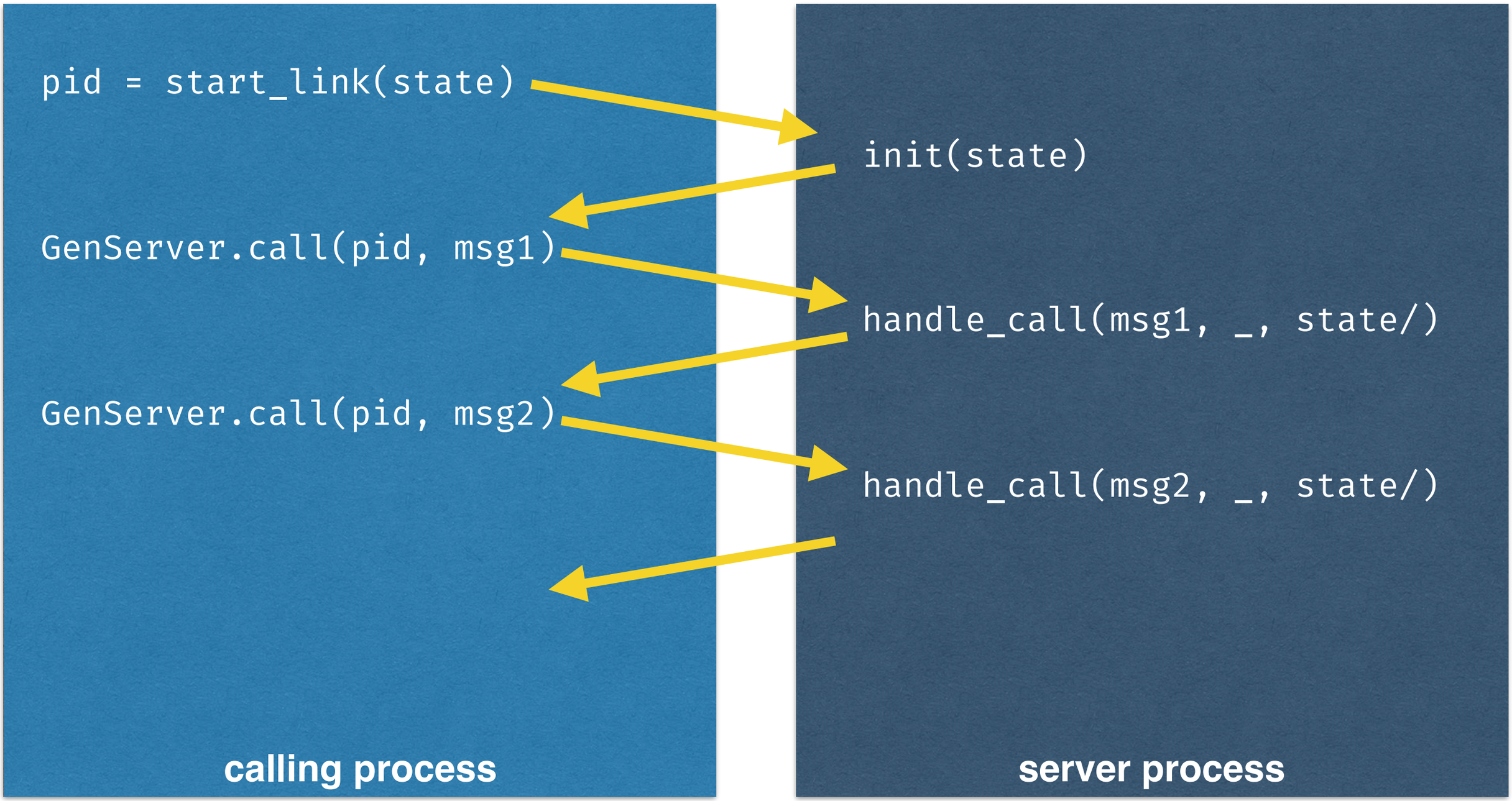
```
def handle_messages(module, state) do
  receive do
    msg ->
      { status, result, new_state } = apply(module, :handle, [ msg, state ])
      if status != :stop do
        handle_messages(module, state)
      else
        result
      end
  end
end

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


pid = start_link(state)
init(state)
GenServer.call(pid, msg1)
handle_call(msg1, _, state/)
GenServer.call(pid, msg2)
handle_call(msg2, _, state/)
calling process
server process

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

```elixir
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 = %{ 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

end
```

API

Implementation
(in server process)

```elixir
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.

```elixir
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

```elixir
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 = %{ 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

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

```elixir
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

```elixir
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
```

Implementation

Public

```elixir
defmodule Summer do

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

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

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

API

Private

```elixir
defmodule Summer.Server do
  use GenServer
  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, valu
    { :reply, result, result }
  end

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

Server

```elixir
defmodule Summer.Implementation do

  def init(total), do: total

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

  def done(total), do: "The total is

end
```

Implementation

Lab

Make the game into a server

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

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 = [
      { Summer,  [0] },
      { Divider, [1] },
    ]

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

Inline Supervisor

```
def start do

    children = [
      { SomeModule, [args] }
    ]

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

Often used in top-level applications

Supervision

- Supervisors do not form part of the logic of the application.

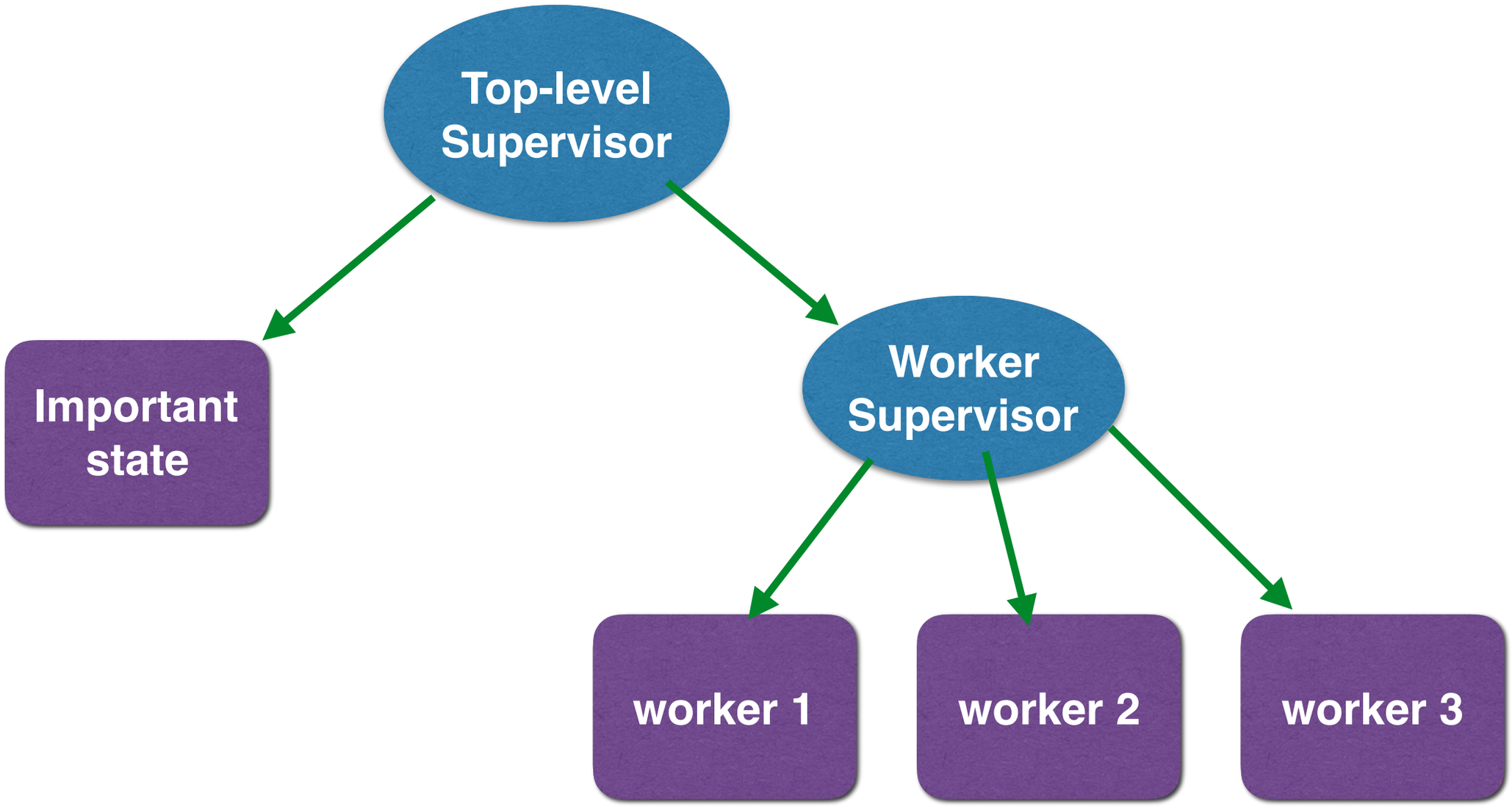
- Supervision structure is separate from the application processes

- You ask a supervisor to start processes, but you then talk directly to those processes

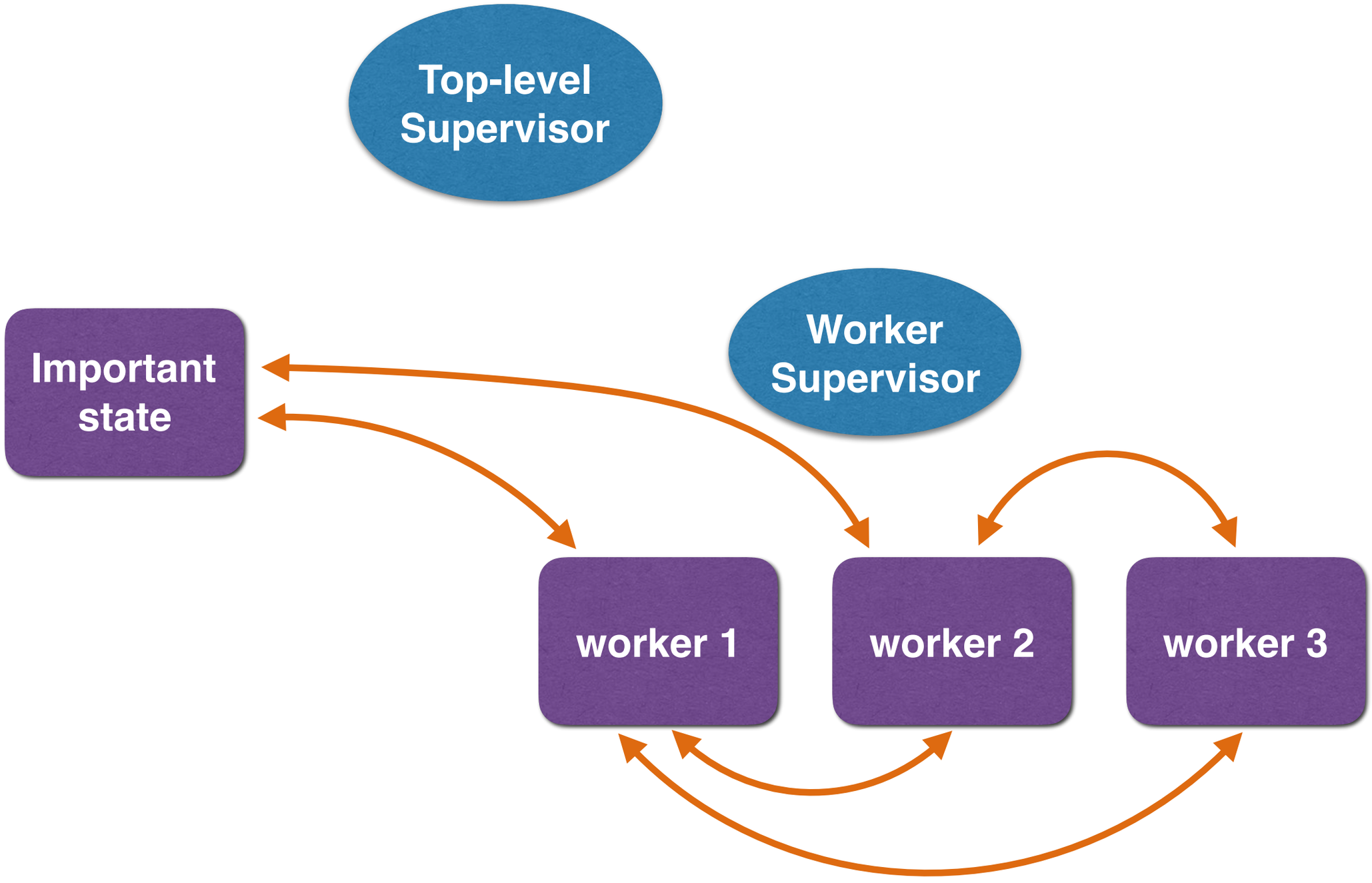
Supervision

- 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



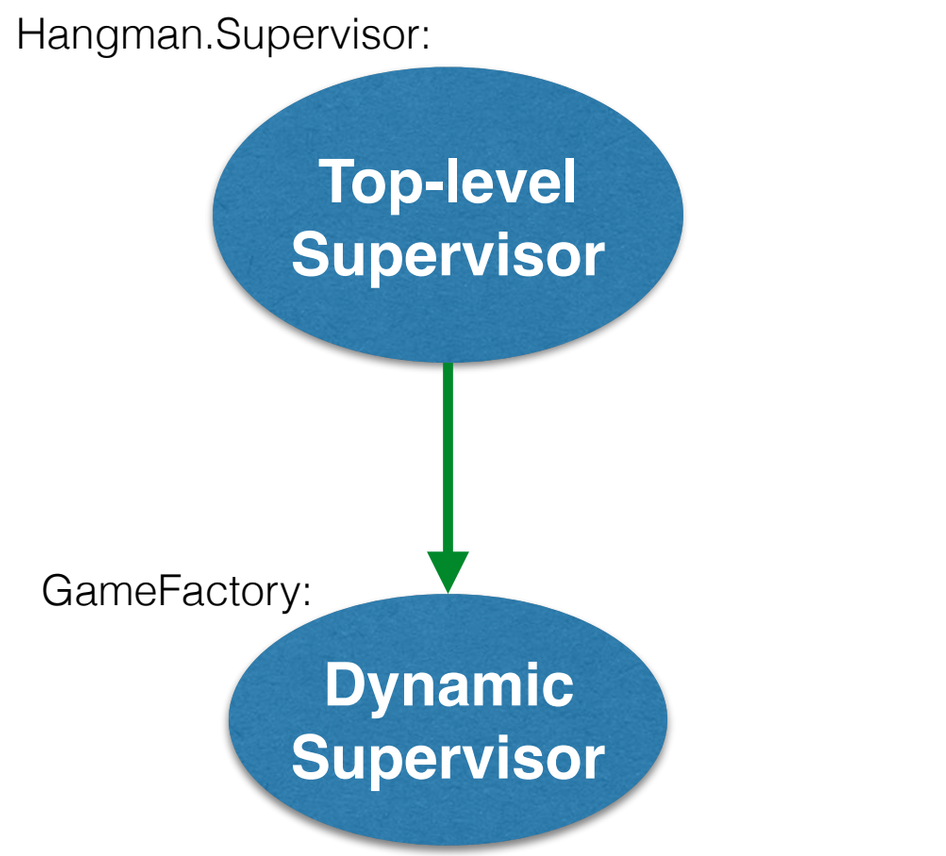
Interaction Structure



Worker Pools

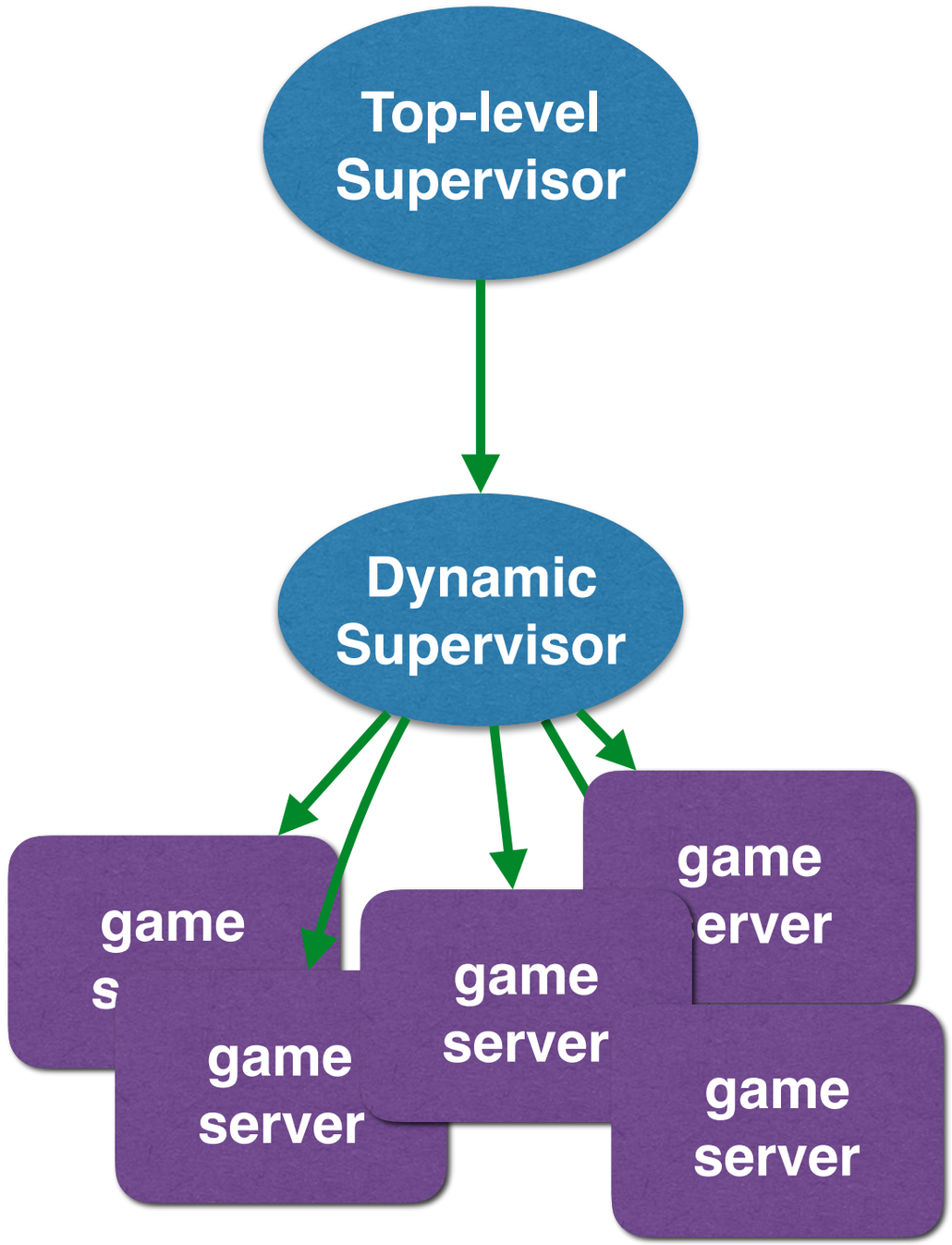
- The list of children you pass to a supervisor is static.
- Dynamic supervisors let you overcome this
- *GenServer factories*

```elixir
defmodule Hangman.Application do
  use Application

  def start(_, _) do

    children = [
      {
        DynamicSupervisor,
        strategy: :one_for_one,
        name:     GameFactory
      }
    ]

    Supervisor.start_link(
      children,
      strategy: :one_for_one,
      name:     Hangman.Supervisor
    )

  end
end
```



```elixir
defmodule Hangman.Application do
  use Application

  def start(_, _) do

    children = [
      {
        DynamicSupervisor,
        strategy: :one_for_one,
        name:     GameFactory
      }
    ]

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

```elixir
defmodule Hangman.GameFactory do
  def create_game(word) do
    DynamicSupervisor.start_child(
      GameFactory,
      { Hangman.Server, word }
    )
  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

```elixir
def start(_type, _args) do

    children = [
      CalcHistorySupervisor, []),
      Summer,
      Subtracter,
      Divider,
      Multiplier,
    ]

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

name of sub supervisor module

```elixir
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, as supplied with Elixir, so you need to tell mix explicitly if you want them started.

Lab 6

- Turn Hangman into a Supervised process (you'll want to use dynamic supervision)
- Make both Hangman and Dictionary into independently started applications

Phoenix

TM

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!

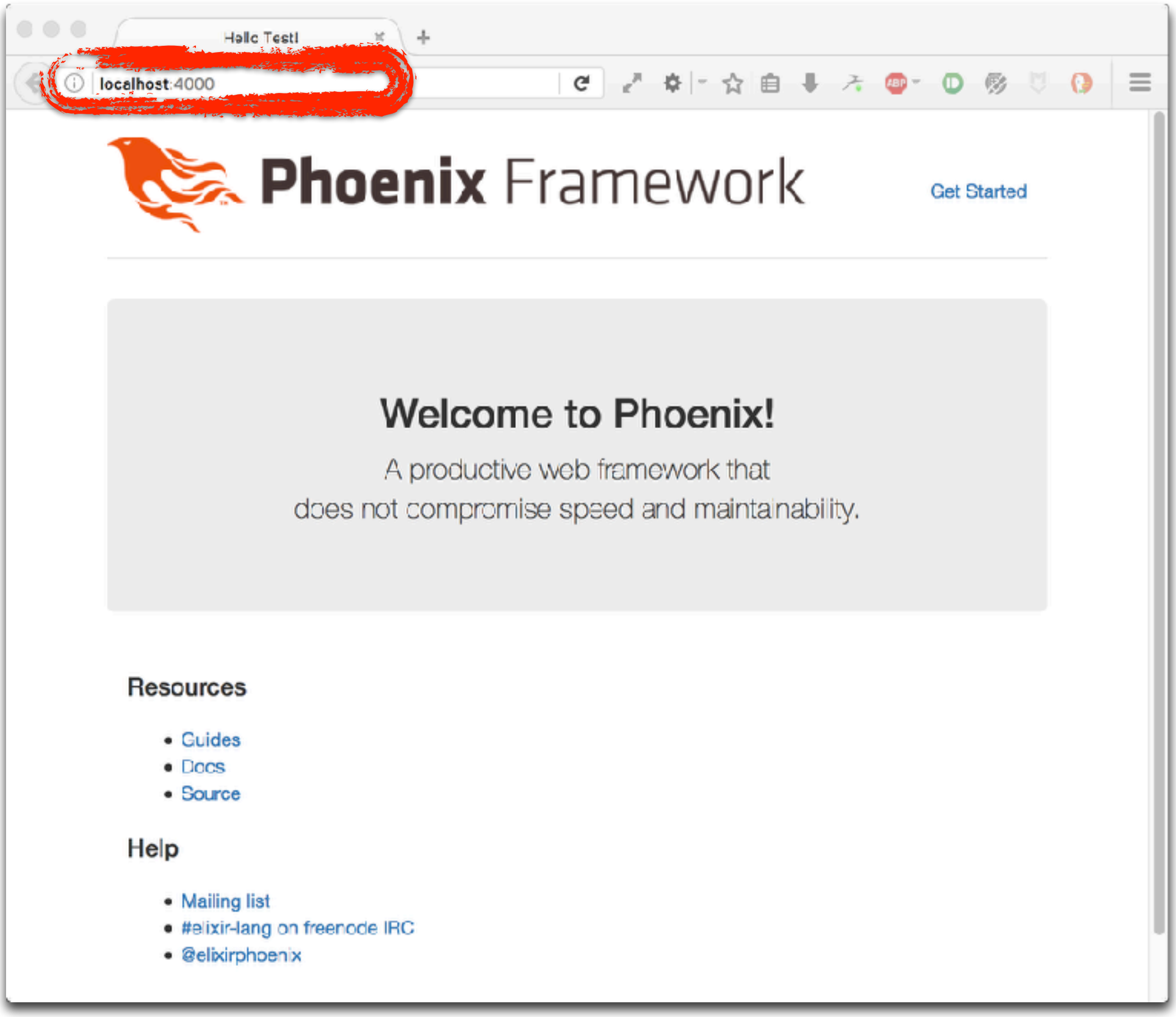
Follow instructions at:

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

tldr;

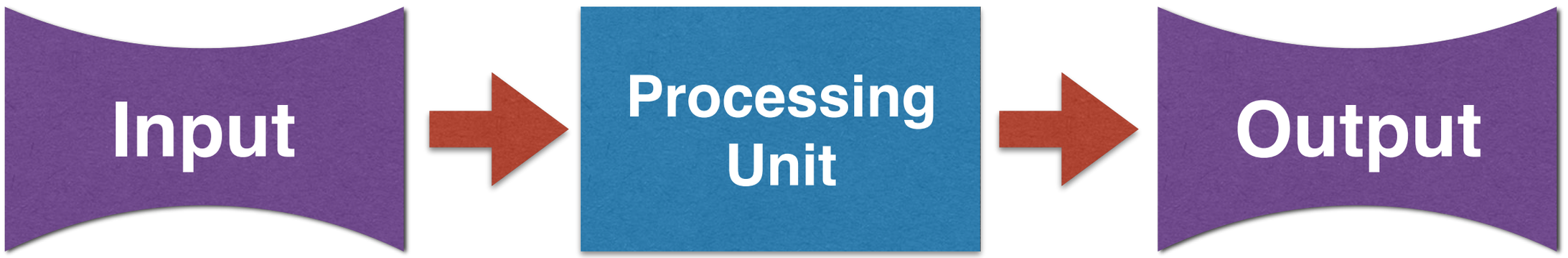
1. mix archive.install https://github.com/phoenixframework/archives/raw/master/phoenix_new.ez
2. Install node.js
3. DO NOT bother with Postgresql
4. Linux users install **inotify**

```
$ 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
```

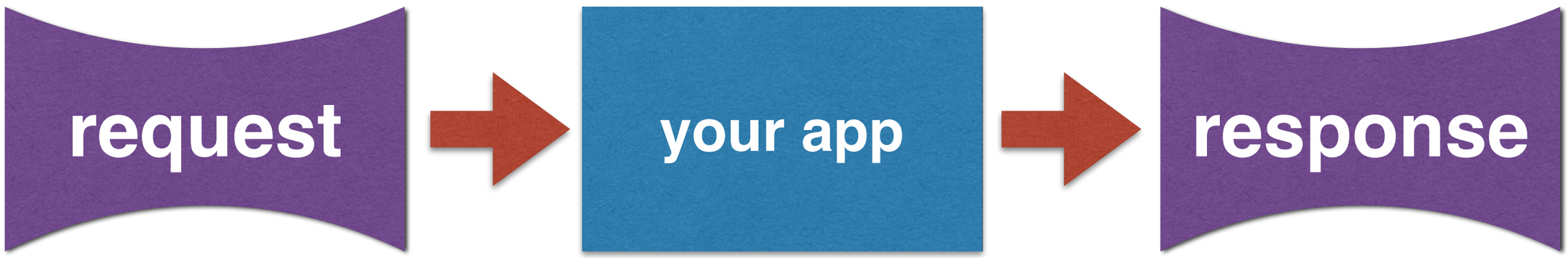

Hello Test!
localhost:4000
Phoenix Framework
Get Started
Welcome to Phoenix!
A productive web framework that
does not compromise speed and maintainability.
Resources
Guides
Docs
Source
Help
Mailing list
#elixir-lang on freenode IRC
@elixirphoenix

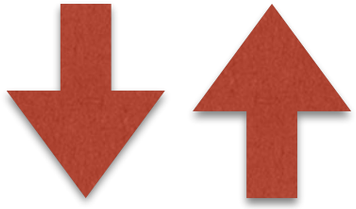
What Phoenix Does

How Computers Work



How Phoenix Works





How it Does It

PLUG

DENMARK


AUSTRALIA

Forbes ranks Happiness

#2

#4

#12

http://www.forbes.com/sites/christopherhelman/2013/01/09/the-worlds-happiest-and-saddest-countries-2/

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

lib/my_app/endpoint.ex

```
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
```

web/router.ex

```elixir
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
```

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

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

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

Yes, this is effectively a plug, too!

web/controllers/page_controller.ex

```elixir
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

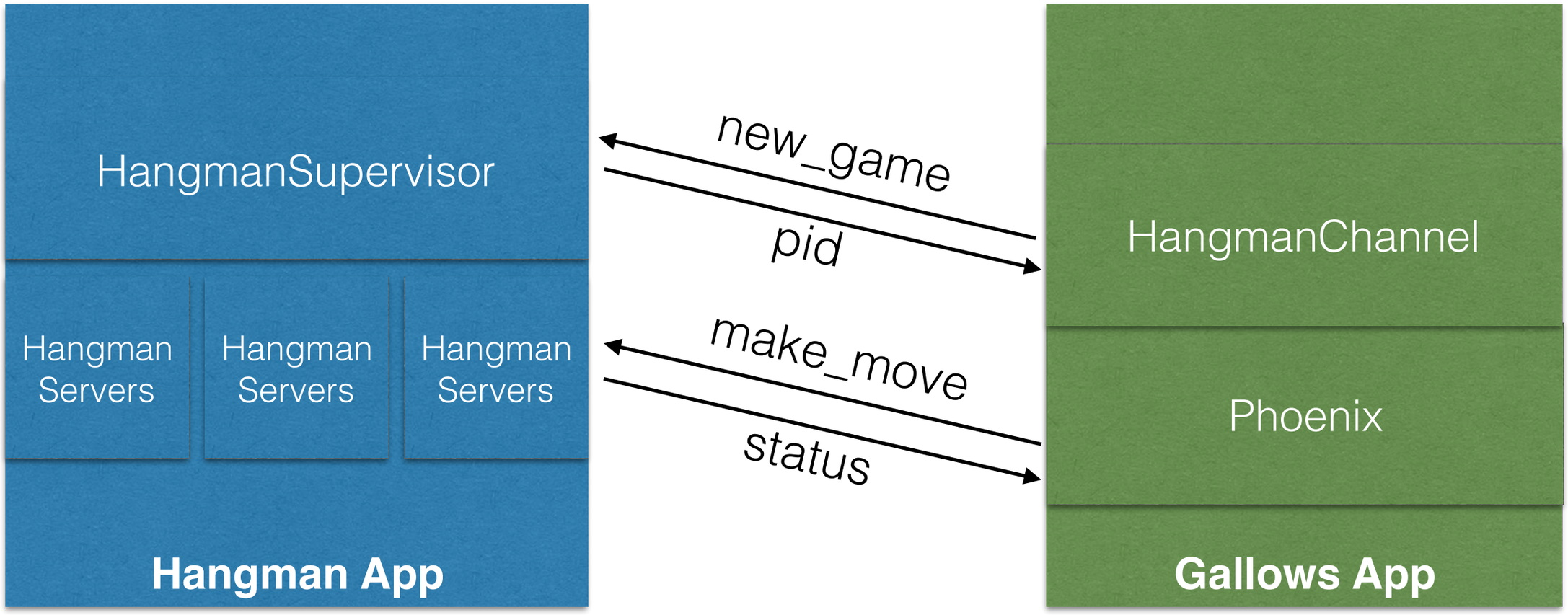
- Client joins a topic on a channel
- Channel controller authorizes
- Either end can send messages
    - channel controller receives client messages via handle_in callback
- Channel controller maintains state in a **socket** structure (passed to all handlers)

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/….


HangmanSupervisor
Hangman Servers
Hangman Servers
Hangman Servers
Hangman App
new_game
pid
make_move
status
HangmanChannel
Phoenix
Gallows App

client Javascript

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

channels/user_socket.ex

```
defmodule Gallows.UserSocket do
  use Phoenix.Socket

  channel "hangman:*", Gallows.HangmanChannel
```

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

```
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
```

client

```
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

```
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 })
```

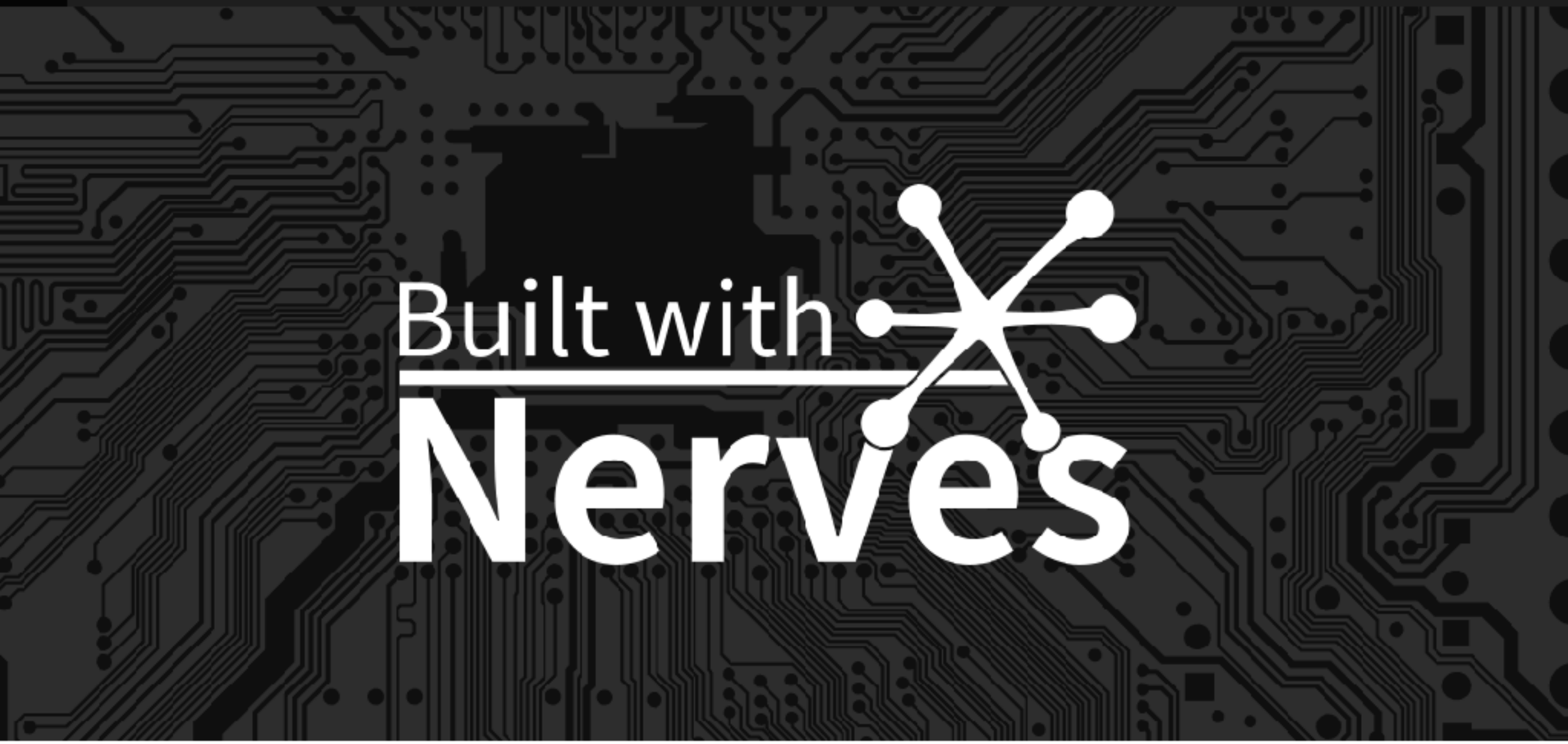
hangman_channel.ex

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

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

But Wait!

There’s more



Nerves

Craft and deploy bulletproof embedded software in Elixir

form

your whole application into as

Framework

Let Nerves take care of the network

Tooling

Go from "mix new" to running code on

Join the community

#nerves channel on Elixir Slack
Nerves on Github
Nerves on the Elixir Forum

Follow @NervesProject
Tweet

Latest News

me1

Scene: Sensor

Tue 4:59 PM

75.0°

Maintenance

"Sensor" is a simple scene that displa
The sensor is in /lib/sensors/temper
The buttons are placeholders showir

me1

Scene: Components

Tue 4:59 PM

Various components

Crash

Event received:

Primary | Success | Info | Light | Warning

Dark | Text | Danger | Secondary

Radio A
Radio B
Radio C

Check Box

Type here...

Choice 1

Password

beast.ex component.ex scene.ex matrix.ex sotb.ex nav.ex

```
omponent.Airships do
nent
```

Shadow of the Beast

FPS mulitplier: 1

Frames: 747
Time: 11.88s
FPS: 62.89

Takeaway

- Programming is transforming state
- Components are good
- Processes ⟺ Objects
- Elixir is a cool hybrid