CONF & COFFEE

LEARN ELIXIR BY BUILDING A REAL TIME CLI CHAT APP

```
git clone
https://github.com/expede/quick_chat.git
```

INTRODUCTIONS 👋

INTRODUCTIONS 👋

WHO AM I?

- Elixir libraries
  - Quark, Algae, Witchcraft
  - Exceptional
  - Operator
- Keynotes
  - ElixirLDN, London
  - Empex, NYC

WHAT WE ARE & *ARE NOT* COVERING

- How Elixir compares to other languages
- Set up a project
- Parts of the core library
- Processes and networking
- No Phoenix
- This is to give you a starting point; you won’t be an expert after 2 hours

HOUSEKEEPING ✅

- Be respectful of each other
- Safe space, zero judgement
- If you ask something that we don't have an answer for,
  we will find the answer for you during a break!
- Does anyone else have questions or ground rules that they'd like
  to raise before we get going?
- *Strongly* encourage to follow along on terminals
- Pair off into exercise buddies

GROUP ACTIVITY

QUICKCHAT DEMO 🚀

https://github.com/expede/quick_chat

A BRIEF HISTORY OF

ERLANG & ELIXIR 

ERLANG

ERLANG

- Agner Krarup Erlang
  & “Ericsson Language”
- High performance 1980s telecom switches
- Prolog-ish syntax
- Same VM as Elixir (BEAM)
- Full introp
- Actor model

```erlang
-module(count_to_ten).
-export([count_to_ten/0]).

count_to_ten() → do_count(0).

do_count(10) → 10;
do_count(N) → do_count(N + 1).
```

ELIXIR

ELIXIR

- Released in 2011
- BDFL José Valim & co
- Several Rails core members
- Runs on Erlang's BEAM VM
- Fully interoperable with Erlang
- Concurrent, fault tolerant, &c

PHILOSOPHY 🤔

- Readability
- Consistency
- Fault tolerance
- Light weight processes
- Modernity (ex. UTF-8 out of the box)

ELIXIR

RUBY INFLUENCE 💎

- Philosophy
  - Friendliness
  - Pragmatism
  - Readability (“like English”)
  - Permissive & dynamic
- Superficial aesthetics
  - **do** blocks
  - Dot syntax
  - Truthiness

BEAM IN THE WILD

- WhatsApp (2 million connections on a single node with low resource usage)
- Amazon
- Facebook
- Motorola
- Ericsson (obviously)
- Heroku
- Riak
- Parallela's Epiphany board

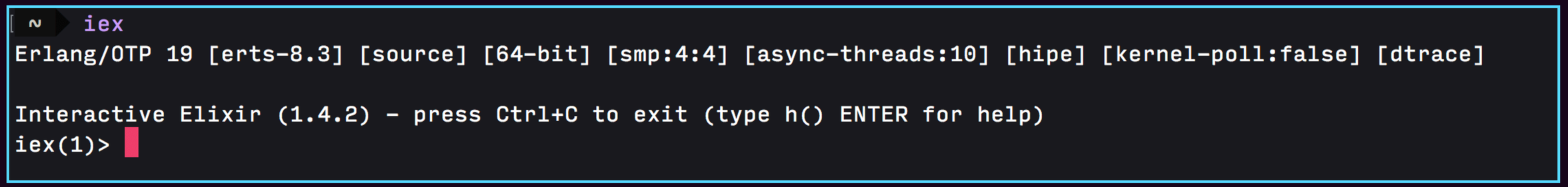
MOTIVATING CASES

- Free lunch is over
- Networking (ie: internet)
- High availability
  - “Nine nines” of uptime
  - 99.9999999%
  - ~32ms/year total downtime(!)
- Soft realtime
- Fault tolerant
- Concurrency
- Distributed computing
- Hot code reloading
- Clustering

BASICS 🏁

BASICS

DIVING IN WITH IEX



BASICS

DIVING IN WITH IEX

```
~ iex
Erlang/OTP 19 [erts-8.3] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false] [dtrace]

Interactive Elixir (1.4.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> 1 + 1
2
iex(2)> "Hello" <> " " <> "World"
"Hello World"
iex(3)>
```

BASICS

IEX BREAK MENU

1. BREAK by `ctl-C`
2. then (a)bort
   - more options in IEx BREAK menu

- (c)ontinue
- (i)nfo
- (p)rocess info
- (l)oaded (modules)
- (k)ill some process(es)
- (D)atabase info & tables
- (d)istribution
- (v)ersion (of BEAM)

BASICS

SYNTAX OVERVIEW

SYNTAX OVERVIEW

“TRUE” PRIMITIVES

```
-2
5

-8.0
1.1

:a
:"so example. very wow."
MyModule
nil
true
false
```

```
<<97, 98, 99, 100, 101, 102>>
"abcdef"

[97, 98, 99, 100, 101, 102]
'abcdef'

{}
{:a}
{-4, 0.9, :a, 'abc', "foo"}

%{a: 1, b: :foo, c: "hi there"}
%{"bar baz" => :quux}

fn x -> x + 1 end
```

BASICS

ANATOMY OF A MODULE

```
# ./foo.ex

defmodule Foo do
  def echo(a), do: IO.inspect(a)
end
```

BASICS

LOADING A MODULE INTO IEX

```
# ./foo.ex

defmodule Foo do
  def echo(a), do: IO.inspect(a)
end
```

```
iex(1)> c("foo.ex")
[Foo]
```

SYNTAX OVERVIEW

TUPLES VS LISTS

TUPLES VS LISTS

TUPLES

- More like arrays
- Contiguous in memory
- Random access
- Constant time access or update `O(1)`

TUPLES VS LISTS

LISTS

- Extremely important datatype
- Unfixed size
- Recursive
- Heterogeneous
- Direct access only to head ("pass the bucket" `O(n)` otherwise)
  - Near-zero cost head access or head append

SYNTAX OVERVIEW

ARITY

```elixir
defmodule WowMath do
  # Anonymous function arity
  # add_anon = fn(x,y) -> x + y end
  # add_anon.(1, 2) # add/0

  def add(), do: 10

  # add/1
  def add(a), do: a + 10

  # add/2
  def add(a, b), do: a + b

  # add/3, despite the default value
  def add(a, b, c \\ 10), do: a + b + c
end
```

SYNTAX OVERVIEW

PIPES

- Reorder the flow of functions
- Linearize function calls for readability
- Goes into the *first* argument position of the next function
- Conceptually there is a “subject”

```
12345
|> Integer.digits() # ⇒ [1, 2, 3, 4, 5]
|> Enum.count() # ⇒ 5
|> Integer.floor_div(3) # ⇒ 1

# Equivalent to
Integer.floor_div(Enum.count(Integer.digits(5)), 3) # ⇒ 1
```

SYNTAX OVERVIEW

"SUBJECT"

- Like an assembly line
  - One subject goes through the pipes
  - Gets changed one function at a time
  - Builds up a bigger computation
    - Composition from the value's perspective

PIPES

OVERUSE

- The following are equivalent:

```
45 |> Integer.to_string()

Integer.to_string(45)
```

SYNTAX OVERVIEW

PATTERN MATCHING & DESTRUCTURING

PATTERN MATCHING & DESTRUCTURING

SIMPLE ASSIGNMENT

```
a = 1
b = %{foo: :bar}
```

PATTERN MATCHING & CONTROL FLOW

MULTIPLE FUNCTION HEADS

```
head =
  fn
    ([])        → nil
    ([h | _]) → h
  end

head.([1, 2, 3])
# ⟹ 1
```

PATTERN MATCHING & CONTROL FLOW

NO ~~ESCAPE~~ RETURN

PATTERN MATCHING & CONTROL FLOW

CASE STATEMENTS

```
case {1, "a", :foo} do
  {_, "a", something} → something
  {2, _,    _} → :two
  _ → :other
end
# ⇒ :foo
```

CASE STATEMENTS

MIXED MATCHING

```elixir
mixed = fn([default: default_status], number_list) ->
  number_list
  |> Enum.count()
  |> Integer.floor_div(10)
  |> case do
       1 -> :one
       2 -> :two
       3 -> :three
       _ -> default_status
     end
end

mixed.([default: :bad], 22..99_999)
```

COND STATEMENTS

ELSE IF

```elixir
if is_bitstring(42) do
  :bitstring
else
  if 1 + 1 == 42 do
    :bad_math
  else
    :default
  end
end
```

```elixir
cond do
  is_bitstring(42) -> :bitstring
  1 + 1 == 42 -> :bad_math
  true -> :default
end
```

FUNCTIONAL PROGRAMMING

DIFFERENT BUT EQUAL

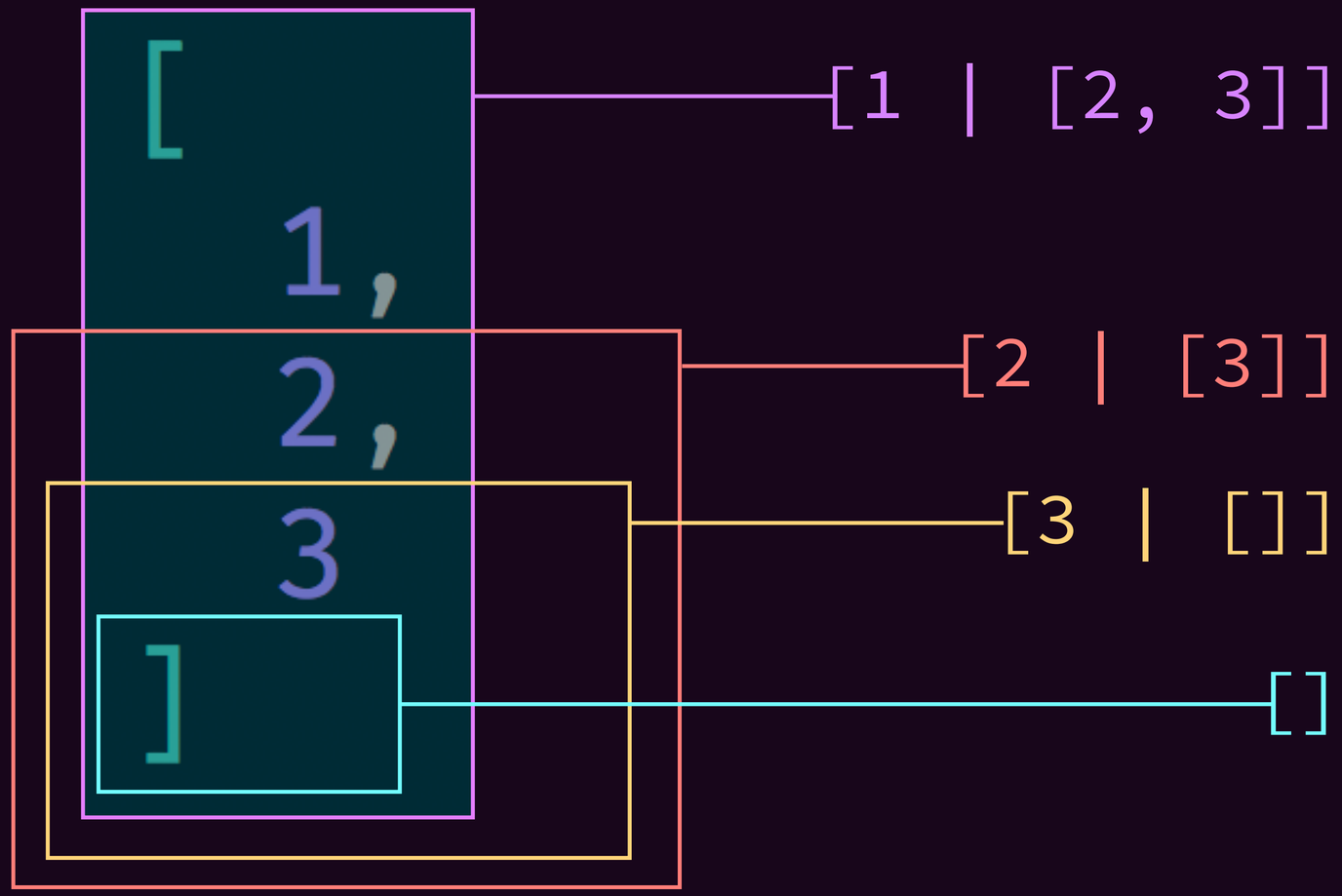
ALAN TURING	ALONZO CHURCH
IMPERATIVE INSTRUCTIONS	EXPRESSIONS, ANALYTIC TRUTHS
GLOBAL STATE "MEMORY"	MANUAL, LOCALIZED STATE "STATELESS"
MECHANISTIC TURING MACHINES	SEMANTIC MATHEMATICAL MODELS

FUNCTIONAL PROGRAMMING

EXPLICIT STATE & RECURSION

EXPLICIT STATE & RECURSION

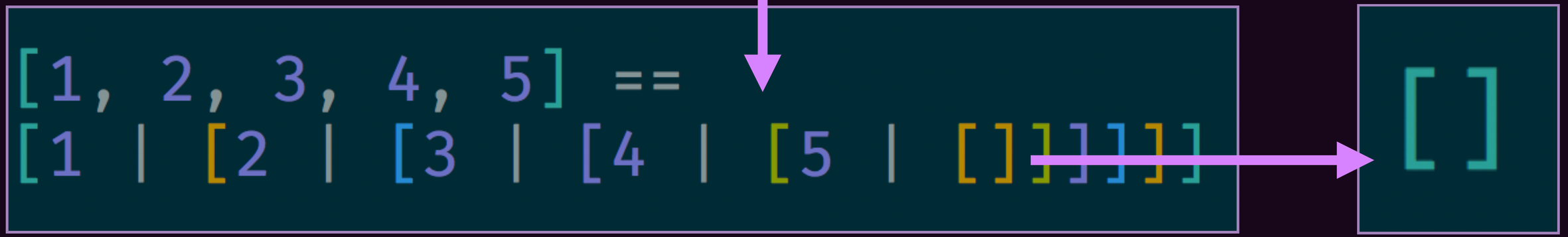
NESTED DATA



EXPLICIT STATE & RECURSION

SOLVING BY INDUCTION

```
sum.([1, 2, 3, 4, 5])
#=> 15
```



EXPLICIT STATE & RECURSION

INDUCTIVE, RECURSIVE SOLUTIONS

```elixir
defmodule Demo.Recursion do
  def sum([]), do: 0
  def sum([value | list]), do: value + sum(list)
end
```

EXPLICIT STATE & RECURSION

INDUCTIVE, RECURSIVE SOLUTIONS

```
[1, 2, 3]
sum([1, 2, 3])
sum([1 | [2, 3]])
1 + sum([2, 3])
1 + sum([2 | [3]])
1 + 2 + sum([3])
1 + 2 + sum([3 | []])
1 + 2 + 3 + sum([])
1 + 2 + 3 + 0
```

```
[1, 2, 3]
sum([1, 2, 3])
sum([1 | [2, 3]])
1 + sum([2, 3])
1 + sum([2 | [3]])
1 + 2 + sum([3])
1 + 2 + sum([3 | []])
1 + 2 + 3 + sum([])
1 + 2 + 3 + 0
```

```elixir
defmodule Demo.Recursion do
  def sum([]), do: 0
  def sum([value | list]), do: value + sum(list)
end
```

A TASTE OF CONCURRENCY

PROCESSES ⚙️

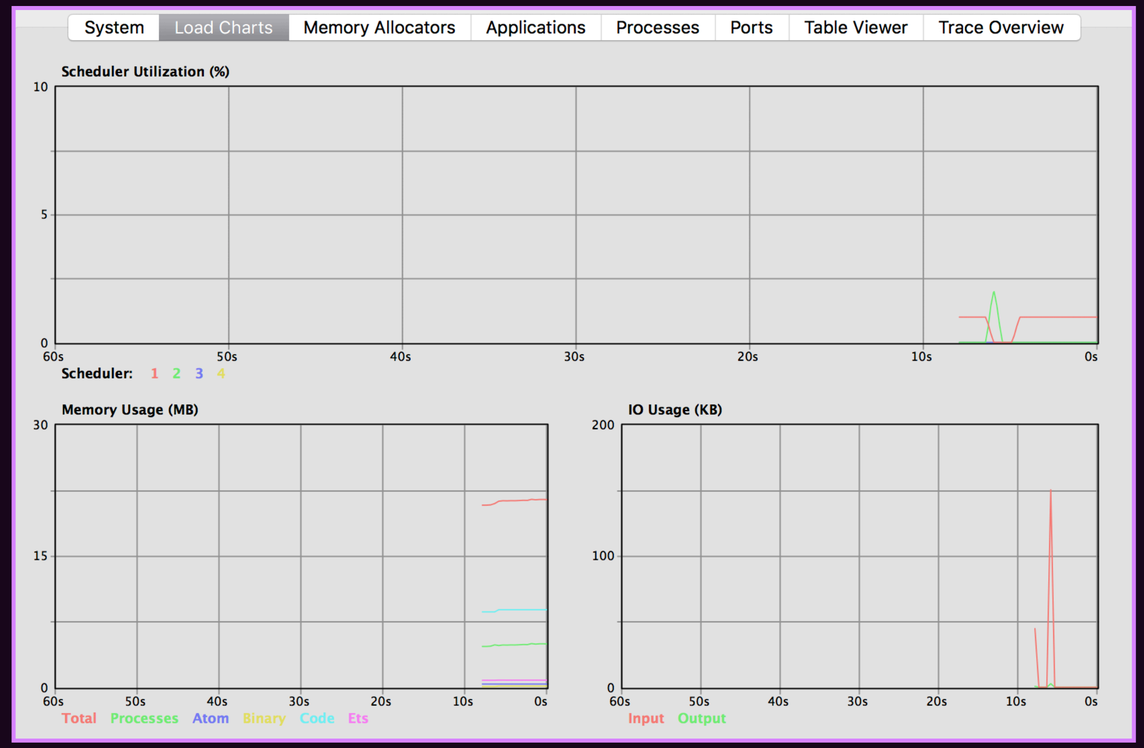
PROCESSES ⚙️

LIGHTWEIGHT

- Lightweight virtual green threads
- Spin up a million processes easily
- spawn ~ fork

PUSHING THE LIMITS

```
iex(8)> :observer.start()
:ok
```



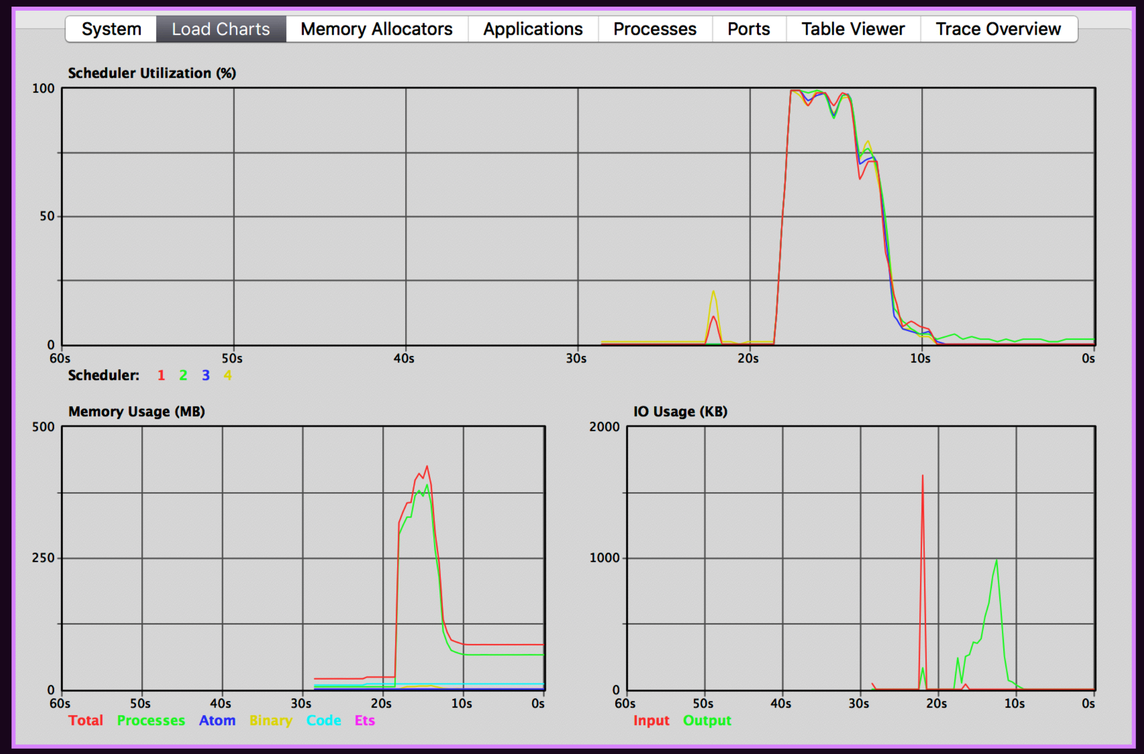
PROCESSES ⚙️

QUICK & DIRTY EXAMPLE

- Do some fake work to do
  - ie: block and keep them in memory
- Do this *many* times
  - 100k concurrent processes, no sweat
- Watch pretty graphs

PUSHING THE LIMITS

```elixir
Enum.each(0..99, fn n →
  spawn(fn →
    Enum.each(0..999, fn m →
      spawn(fn →
        name = "⚙ #{(n * 100) + m}"
        IO.puts "    Start  #{name}"

        time =
          [10, 500, 1000, 2000, 5000]
          ▷ Enum.random()
          ▷ :rand.uniform()

        Process.sleep(time)

        IO.puts "    Finish #{name} in     #{time}ms"
      end)
    end)
  end)
end)
```



ERROR HANDLING

TAGGED TUPLES

TAGGED TUPLES

`{:ok, “great”}`

- Generally comes in the form `{:status, value}`
  - `{:ok, 42}`, `{:error, :bad_val}`, `{:error, :bad_val, 42}`
- Signify some state in the first position
- Extremely lightweight
- Erlang convention (used all over the place)
- Can be as long or short as you want
  - Could we use other tags than `:ok` and `:error`?
  - Can you think of any gotchas?

LET’S WRITE QUICKCHAT 💪🏻

WHAT ARE WE WRITING?

GAME PLAN 🤔

GAME PLAN 🤔

FUNCTIONALITY

- Broadcast to everyone
- Send direct messages
- See a list of everyone that you’re directly connected to
- Forward messages on to others

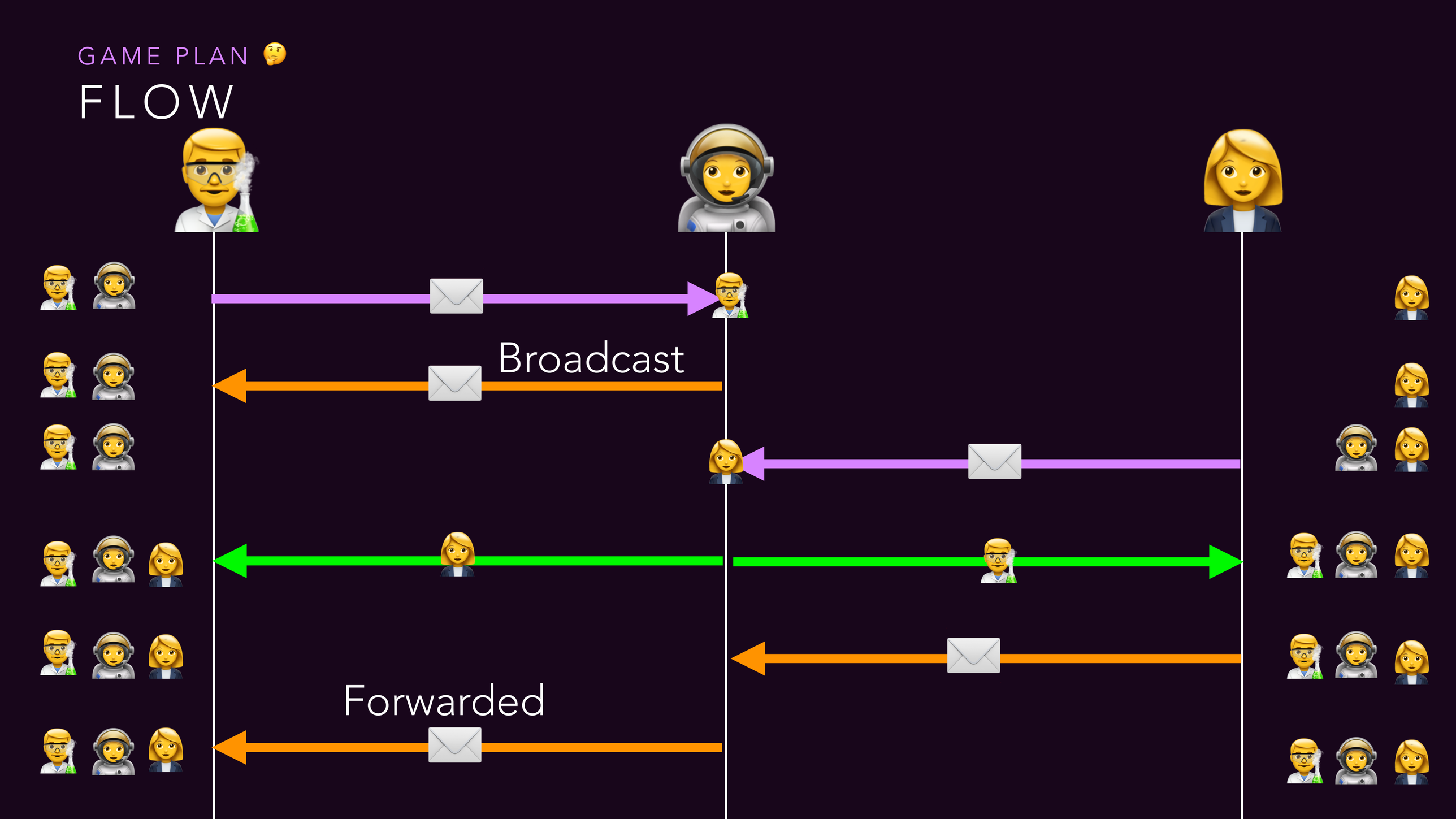
GAME PLAN 🤔

INTERNALS

- For simplicity, we're just printing messages
  - Not keeping the history
- Keep a list of known peers
- When sending or receiving a DM, record the address
  - If you get a DM from a new peer
    - Forward their address to all peers
- When sending a public broadcast
  - Print message
  - Send to all peers with a unique ID
- When receiving a new public broadcast (by ID)
  - Print message
  - Forward to all peers
- If receiving a known public message
  - Do nothing

GAME PLAN 🤔

FLOW

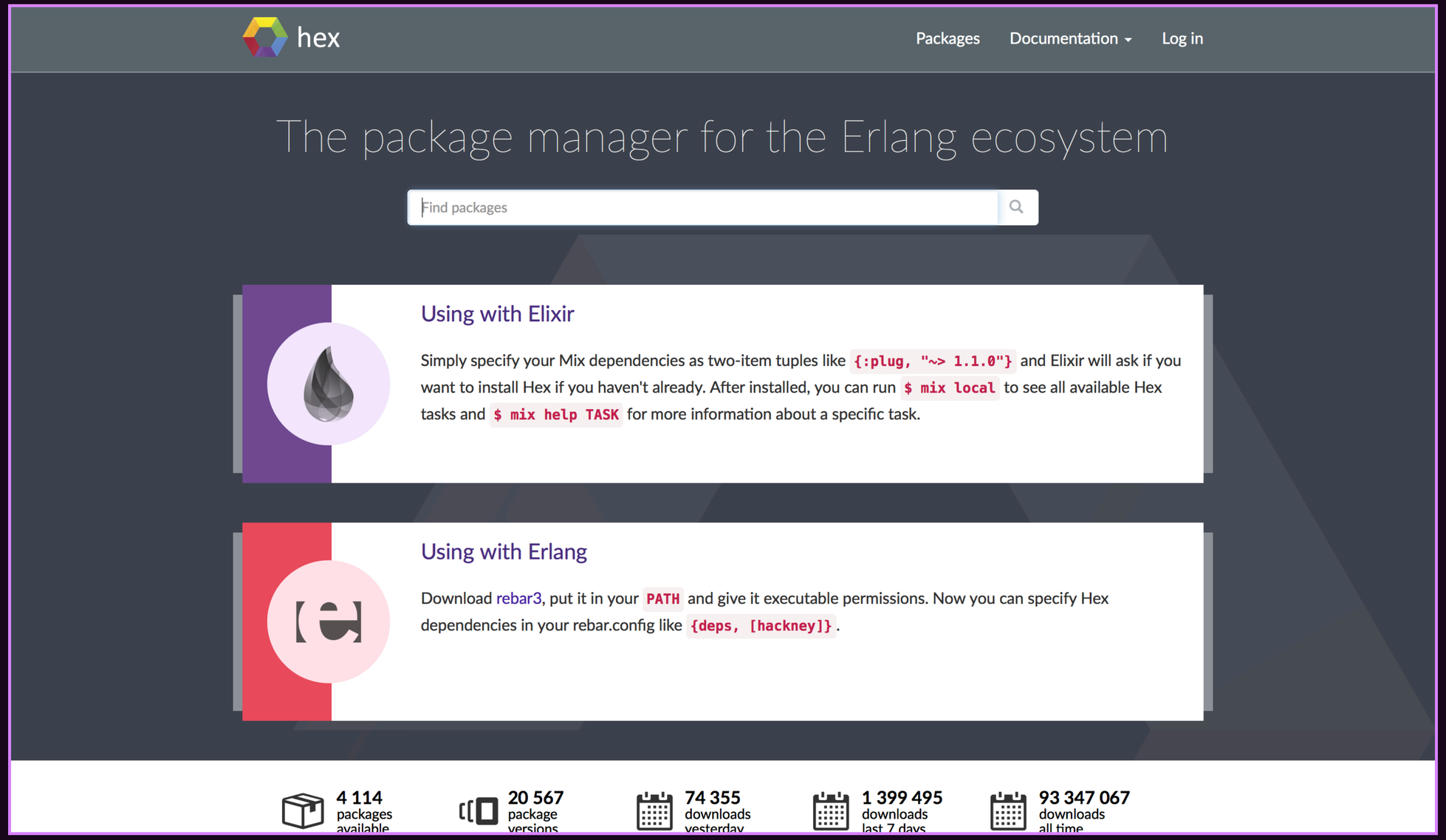


MIXING UP A

PROJECT 

PROJECT 🍹

LET'S TALK ABOUT HEX, BABY 🎶



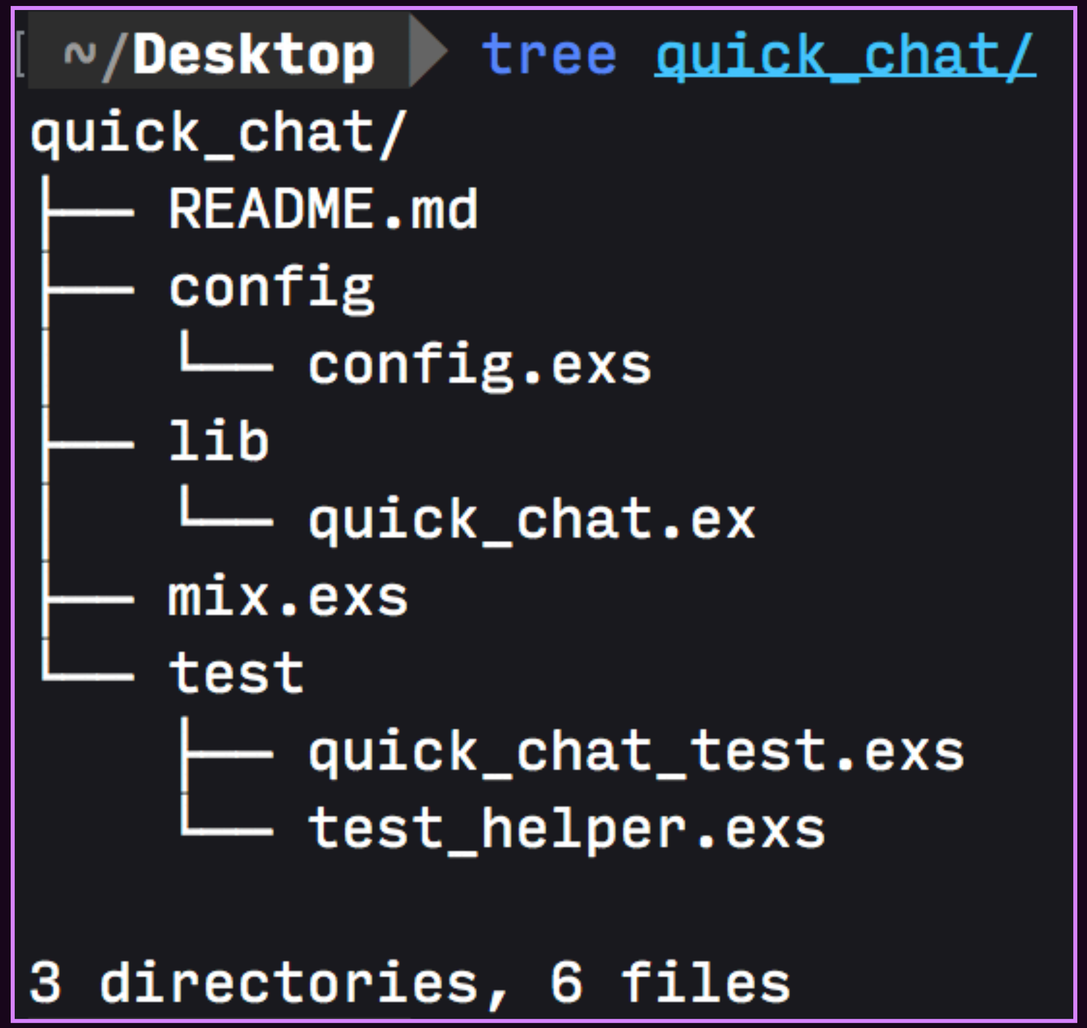
PROJECT 🍹

MIX NEW

```
~/Desktop ▶ mix new quick_chat                    496ms
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/quick_chat.ex
* creating test
* creating test/test_helper.exs
* creating test/quick_chat_test.exs

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

    cd quick_chat
    mix test

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

PROJECT 🍹

ANATOMY OF A PROJECT



PROJECT 🍹

MIXFILE

```elixir
defmodule QuickChat.MixProject do
  use Mix.Project

  def project do
    [
      app: :quick_chat,
      version: "0.1.0",
      elixir: "~> 1.6",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end

  # Run "mix help compile.app" to learn about applications.
  def application do
    [
      extra_applications: [:logger]
    ]
  end

  # Run "mix help deps" to learn about dependencies.
  defp deps do
    [
      # {:dep_from_hexpm, "~> 0.3.0"},
      # {:dep_from_git, git: "https://github.com/elixir-lang
    ]
  end
end
```

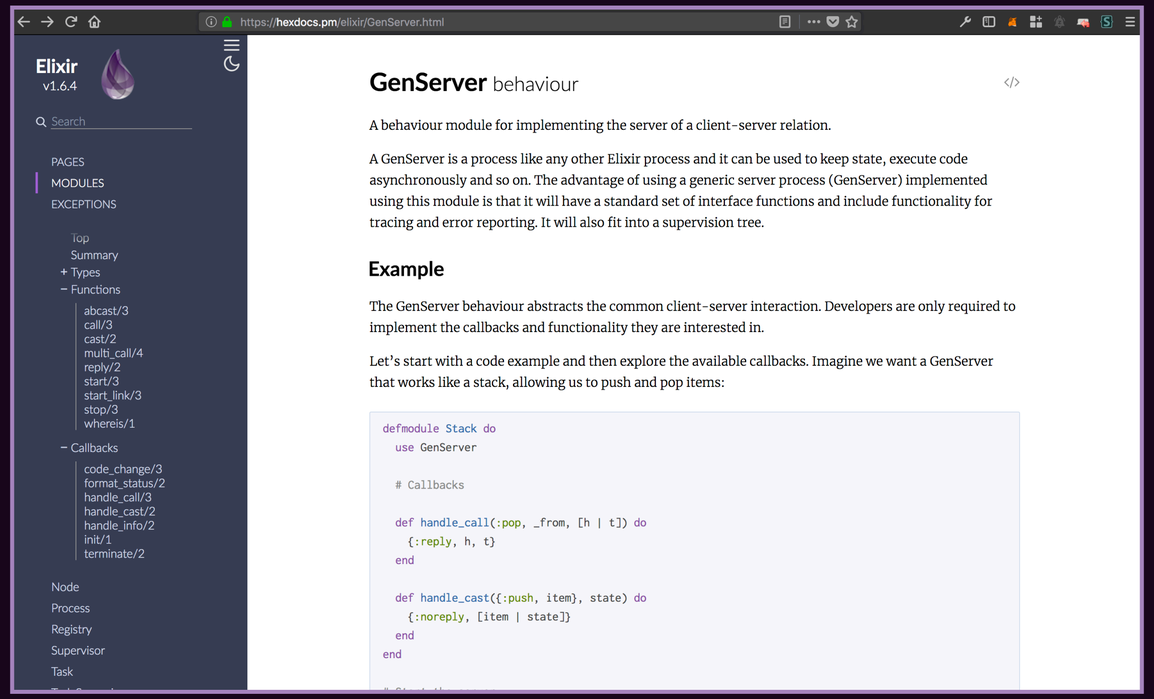
PROJECT 🍹

COMPILE & RUN

```
~/D/quick_chat ▶ mix compile
Compiling 1 file (.ex)
Generated quick_chat app
~/D/quick_chat ▶ iex -S mix
Erlang/OTP 20 [erts-9.3] [source]

Interactive Elixir (1.6.4) - press
iex(1)> QuickChat.hello
:world
iex(2)>
```

PROJECT 🍹

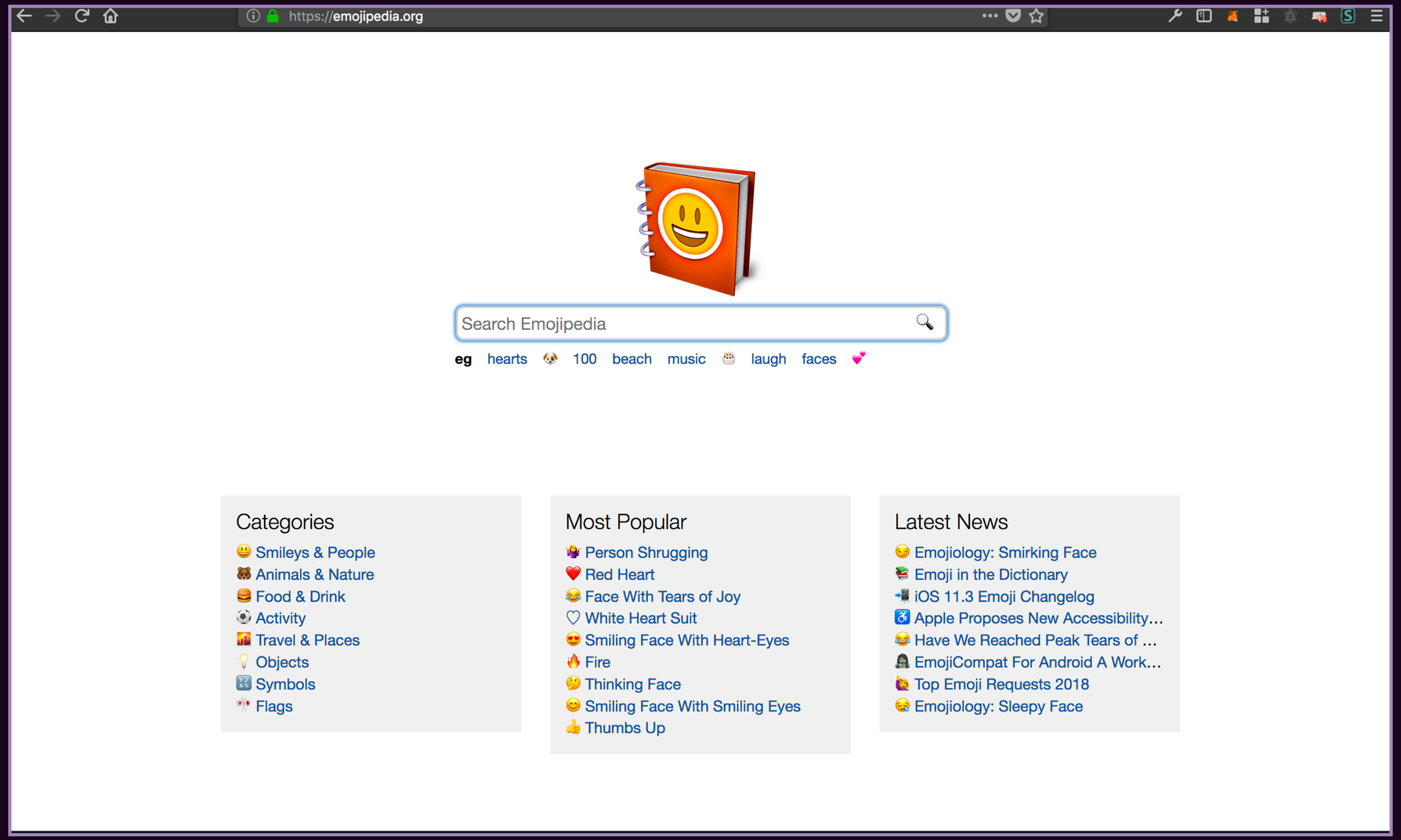
STANDARD LIB DOCS




```
iex(4)> h GenServer.start_link/3

                def start_link(module, args, options \\ [])

    @spec start_link(module(), any(), options()) :: on_start()

Starts a GenServer process linked to the current process.

This is often used to start the GenServer as part of a supervision tree.

Once the server is started, the c:init/1 function of the given module is called
with args as its arguments to initialize the server. To ensure a synchronized
```

PROJECT 🍹

EMOJIPEDIA.ORG



OTP

GENERIC SERVER 🤝

GENERIC SERVER 🤝

A LITTLE DATABASE EXAMPLE

```elixir
defmodule MyDB do
  use GenServer

  # ========= #
  # Client API #
  # ========= #

  def new, do: GenServer.start_link(__MODULE__, :ok)

  def set(pid, key, value), do: GenServer.cast(pid, {:set, key, value})

  def get(pid, key), do: GenServer.call(pid, {:get, key})
  def dump(pid), do: GenServer.call(pid, {:get_all})

  # ========= #
  # Server API #
  # ========= #

  def init(:ok), do: {:ok, %{}}

  def handle_call({:get_all}, _from, store) do
    {:reply, store, store}
  end

  def handle_call({:get, key}, _from, store) do
    {:reply, Map.get(store, key), store}
  end

  def handle_cast({:set, key, value}, store) do
    {:noreply, Map.put(store, key, value)}
  end
end
```

GENERIC SERVER 🤝

INTERFACE ⏺🔀⏯

GENERIC SERVER 🤝

`init(input)`

- Called by the framework on `start/3` and `start_link/3`
- Must return `{:ok, initial_state}` or `:ignore` to continue
  - State may just be `nil` if it's fully stateless

```elixir
def init(:ok), do: {:ok, %{}}
```

GENERIC SERVER 🤝

`handle_call(request, from, state)`

- Synchronous & blocking
- Includes who the caller was
- Usually returns `{:reply, reply, new_state}`
- May set a timeout for next message
  - `{:reply, reply, new_state, timeout}`
  - If times out, calls `handle_info/2`

```elixir
def handle_call({:get_all}, _from, store) do
  {:reply, store, store}
end

def handle_call({:get, key}, _from, store) do
  {:reply, Map.get(store, key), store}
end
```

GENERIC SERVER 🤝

handle_cast(request, state)

- Asynchronous
- {:noreply, new_state}
- {:stop, reason, new_state}

```elixir
def handle_cast({:set, key, value}, store) do
  {:noreply, Map.put(store, key, value)}
end
```

GENERIC SERVER 🤝

`handle_info(msg, state)`

- Handles everything else
- Same messages as `handle_cast/2`

GENERIC SERVER 🤝

NICER API

```elixir
def new, do: GenServer.start_link(__MODULE__, :ok)

def set(pid, key, value), do: GenServer.cast(pid, {:set, key, value})

def get(pid, key), do: GenServer.call(pid, {:get, key})
def dump(pid), do: GenServer.call(pid, {:get_all})
```

GENERIC SERVER 🤝

PLAY WITH THE DB

```
~/D/W/R/C/E/up_run  master $…  ism
Erlang/OTP 20 [erts-9.3] [source] [64-bit] [sm
:10] [hipe] [kernel-poll:false] [dtrace]

Interactive Elixir (1.6.4) - press Ctrl+C to e
iex(1)> alias UpRun.MyDB
UpRun.MyDB
iex(2)> {:ok, db} = MyDB.new()
{:ok, #PID<0.112.0>}
iex(3)> UpRun.MyDB.dump(db)
%{}
iex(4)> MyDB.set(db, :hi, 42)
:ok
iex(5)> UpRun.MyDB.dump(db)
%{hi: 42}
iex(6)> UpRun.MyDB.get(db, :hi)
42
iex(7)> UpRun.MyDB.set(db, "welp", [1,2,3])
:ok
iex(8)> UpRun.MyDB.dump(db)
%{:hi => 42, "welp" => [1, 2, 3]}
iex(9)>
```

GENERIC SERVER 🤝

TAKE HOME EXERCISE

- “ScanDB”
- Same as the example, except:
  - Keep all previous values in a list
  - ```
    ScanDB.push(db, :a, 1)
    ScanDB.push(db, :a, 2)
    ScanDB.push(db, :a, 3)
    ScanDB.dump(db)
    #=> %{a: [3, 2, 1]}
    ```
    ```

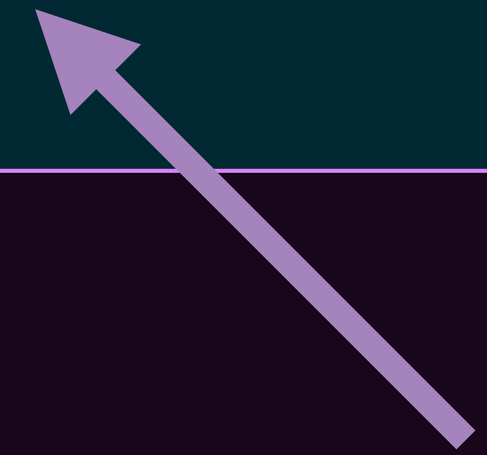
QUICK CHAT

SET UP

SET UP

MODULE

```elixir
defmodule QuickChat.Server do
  use GenServer
end
```



SET UP

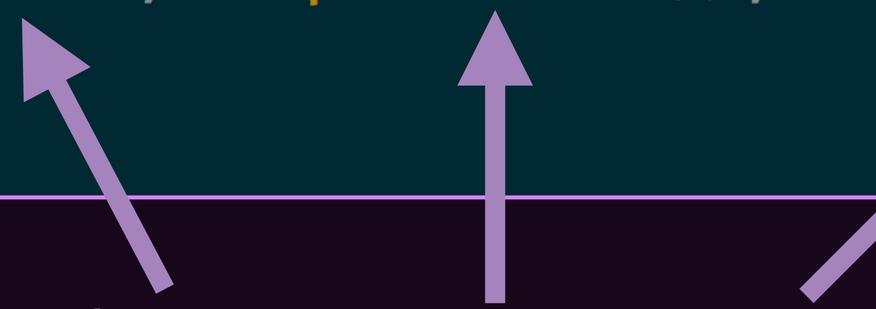
INIT/1

```elixir
defmodule QuickChat.Server do
  use GenServer

  def init([owner]) do
  end
end
```

SET UP

START IT UP!

```elixir
defmodule QuickChat.Server do
  use GenServer

  def init([owner]) do
    IO.puts("Connected as #{Node.self()}")
    {:ok, {owner, MapSet.new(), MapSet.new()}}
  end
end
```



{me, nonces, peers}

ie: our state model (session)

SET UP

START IT UP

```
[ ~/D/V/quick_chat  master *  iex -S mix
Erlang/OTP 20 [erts-9.3] [source] [64-bit] [smp:4:4] |

Interactive Elixir (1.6.4) - press Ctrl+C to exit (typ
[iex(1)> GenServer.start(QuickChat.Server, [self()])
Connected as nonode@nohost
{:ok, #PID<0.111.0>}
iex(2)>
```

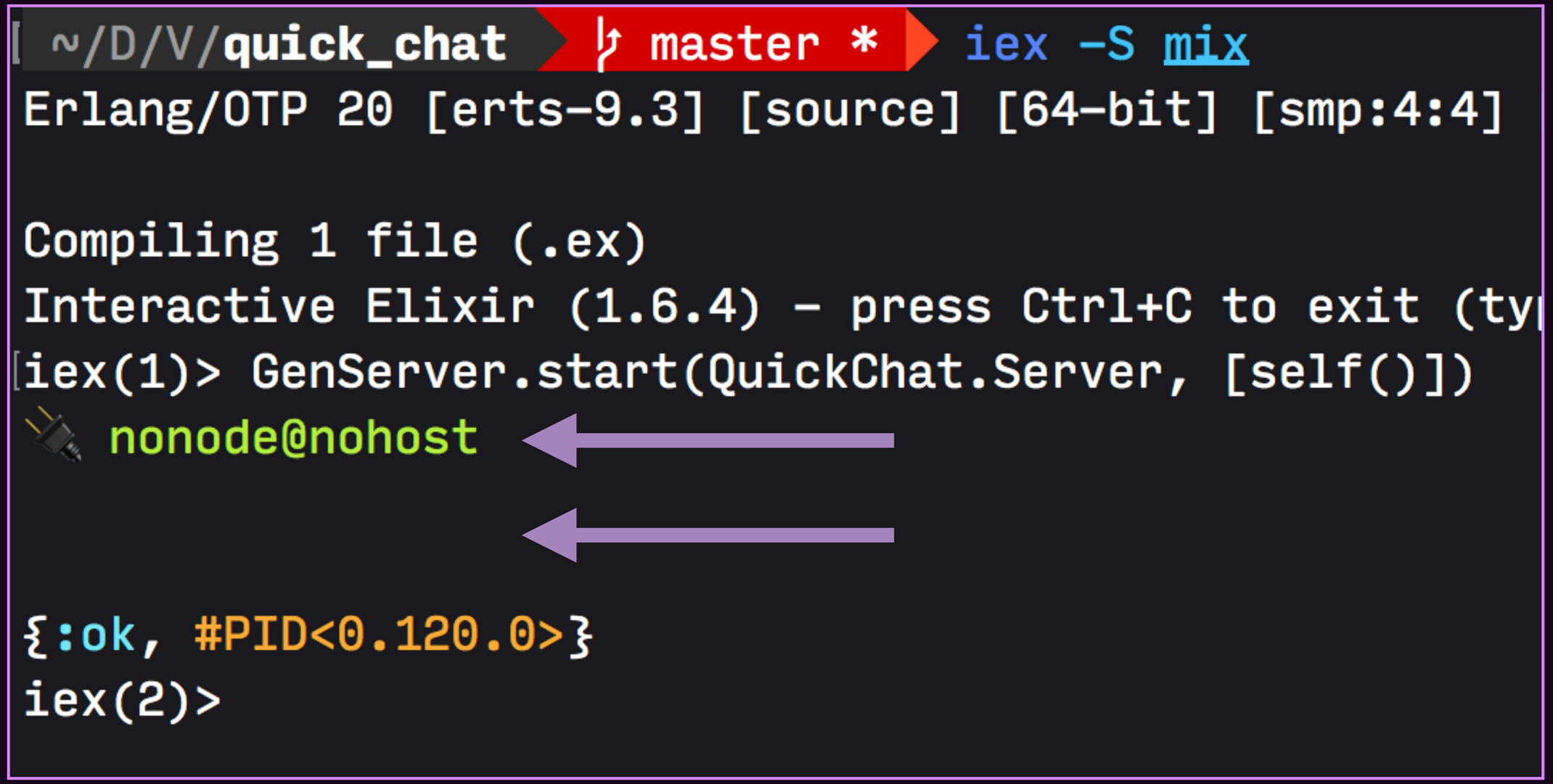
SET UP

START IT UP!

```elixir
def init([owner]) do
  log(nil, Node.self(), "🖧", :green)
  {:ok, {owner, MapSet.new(), MapSet.new()}}
end

def log(text, sender, icon, colour \\ :white) do
  [colour, "#{icon} #{sender}\n#{text}\n"]
  |> IO.ANSI.format(true)
  |> IO.puts()
end
```

SET UP

MAKE PRETTIER MESSAGES

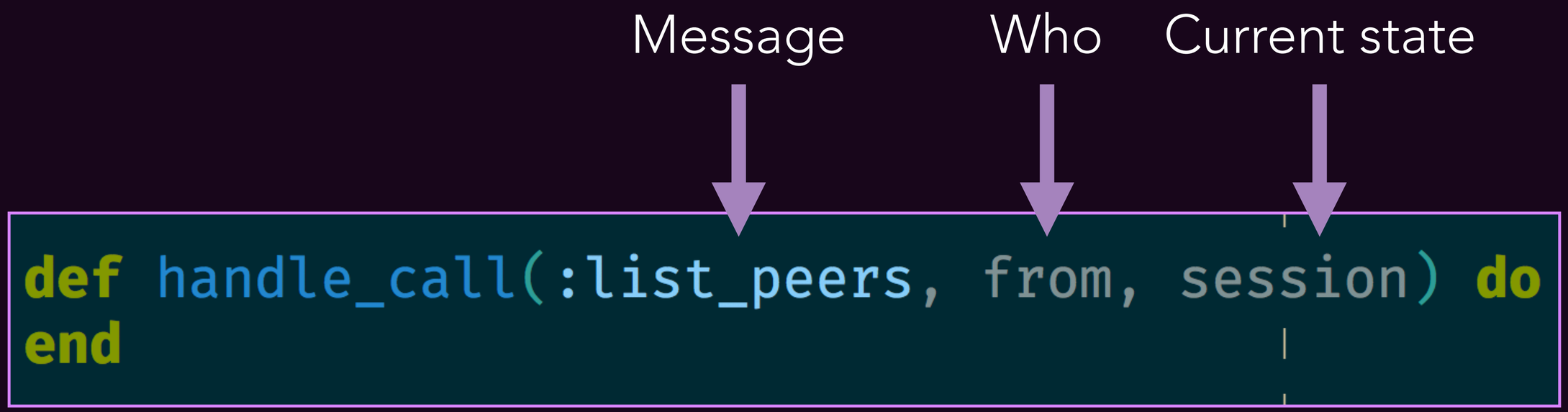


QUICK CHAT

REQUEST DATA

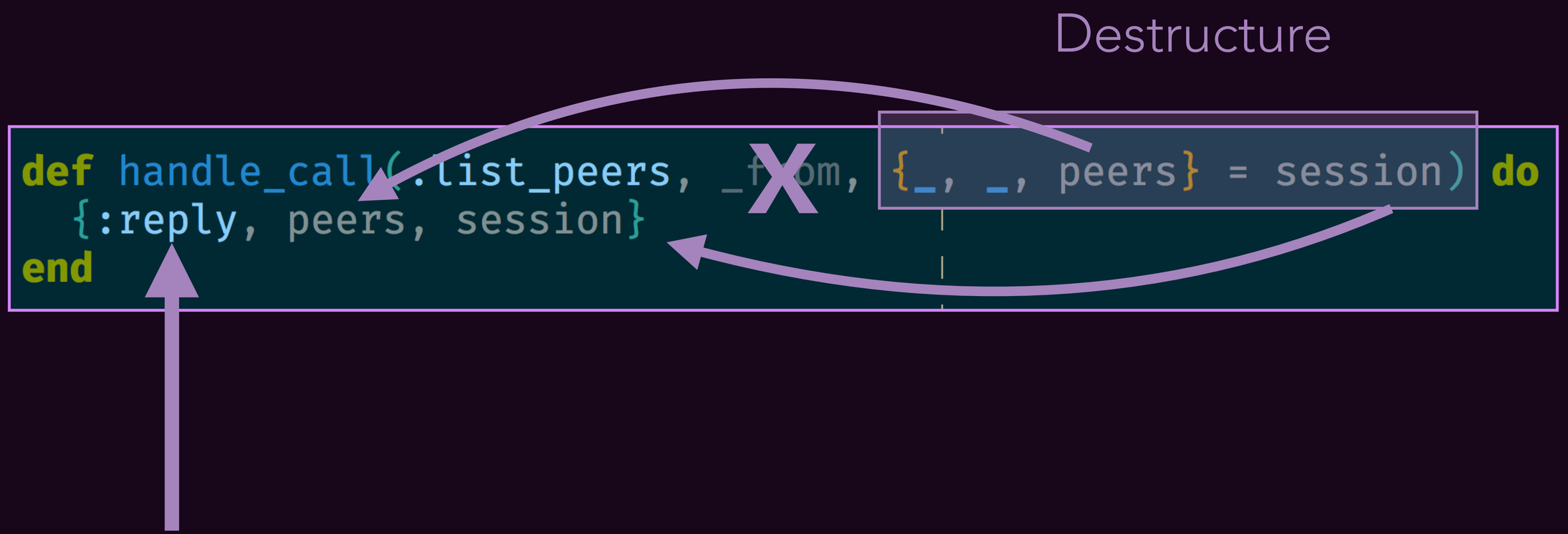
REQUEST DATA

HANDLE_CALL/3



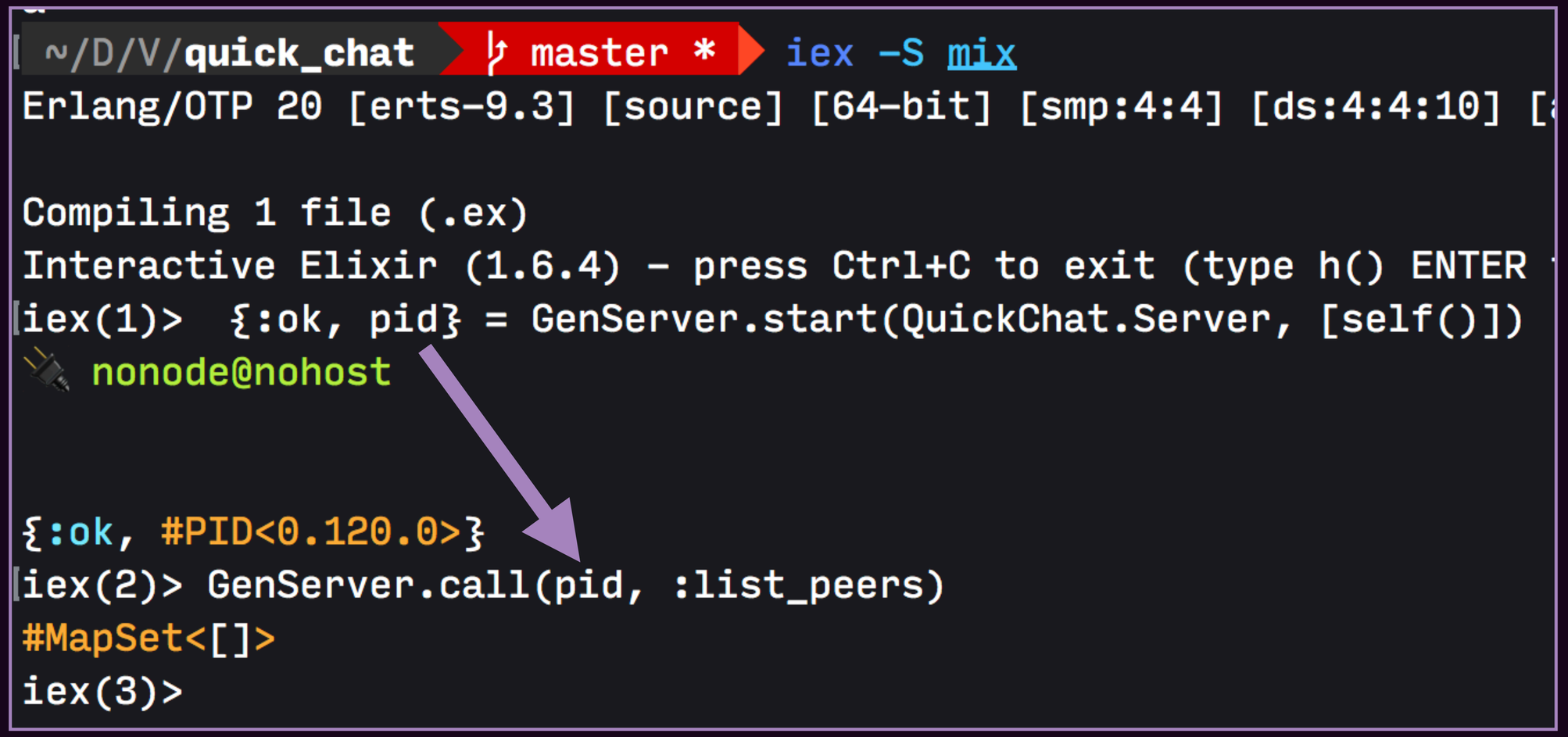
REQUEST DATA

HANDLE_CALL/3



REQUEST DATA

CALL/2



REQUEST DATA

HANDLE_CALL/3

```elixir
def handle_call(:list_peers, _from, {_, _, peers} = session) do
  {:reply, MapSet.to_list(peers), session}
end
```

```
iex(4)> GenServer.call(pid, :list_peers)
[]
```

QUICK CHAT

PUBLIC MESSAGES

PUBLIC MESSAGES

HANDLE_CAST/2

```elixir
def handle_cast({:msg, peer, nonce, text} = msg, {owner, nonces, peers}) do
  log(text, peer, "😁")
  {:noreply, {owner, MapSet.put(nonces, nonce), peers}}
end
```

…but we also want to forward to our peers

QUICK CHAT

FORWARDING

FORWARDING

HANDLE_CAST/2

```elixir
def handle_cast({:msg, peer, nonce, text} = msg, {owner, nonces, peers}) do
  log(text, peer, "😁")
  forward(peers, msg)
  {:noreply, {owner, MapSet.put(nonces, nonce), peers}}
end
```

```elixir
def forward(peers, payload) do
  Enum.each(peers, fn peer ->
    peer
    |> address()
    |> GenServer.cast(payload)
  end)
end
```

FORWARDING

HANDLE_CAST/2

```elixir
def forward(peers, payload) do
  Enum.each(peers, fn peer ->
    peer
    |> address()
    |> GenServer.cast(payload)
  end)
end
```

```elixir
def address(room) when is_bitstring(room) do
  room
  |> String.to_atom()
  |> address()
end

def address(room), do: {:chat, room}
```

QUICK CHAT

DEDUPING

DEDUPING

HANDLE_CAST/2

```elixir
def handle_cast({:msg, peer, nonce, text} = msg, {owner, nonces, peers} = session) do
  if MapSet.member?(nonces, nonce) do
    {:noreply, session}
  else
    log(text, peer, "😁")
    forward(peers, msg)
    {:noreply, {owner, MapSet.put(nonces, nonce), peers}}
  end
end
```

QUICK CHAT

BROADCAST COMMAND

BROADCAST COMMAND

HANDLE_CAST/2

Why?

```elixir
def handle_call({:broadcast, text}, {from, _}, {owner, nonces, peers}) when from == owner do
  log(text, "Me", "😎", :yellow)

  new_nonce = nonce()
  forward(peers, {:msg, me(), new_nonce, text})

  {:reply, :ok, {owner, MapSet.put(nonces, new_nonce), peers}}
end

def me, do: to_string(node())

def nonce, do: :crypto.strong_rand_bytes(8)
```

Look familiar?

IEX

HANDLE_CALL/3

Cheating a bit for demo

```
iex(1)> GenServer.start(QuickChat.Server, [self()], name: :chat)
🔌 nonode@nohost


{:ok, #PID<0.120.0>}
iex(2)> GenServer.call(:chat, {:broadcast, "hello"})
😎 Me
hello

:ok
iex(3)> GenServer.call(:chat, {:broadcast, "bye"})
😎 Me
bye

:ok
```

QUICK CHAT

ADDING PEERS

ADDING PEERS

HANDLE_CAST/2

```elixir
def handle_cast({:add_peers, external}, {owner, nonces, internal} = session) do
  external
  |> MapSet.difference(internal)
  |> MapSet.size()
  |> case do
    0 ->
      {:noreply, session}

    count ->
      log("Attached to #{count} new peer(s)", "Alert", "🤝", :green) |

      {:noreply, {owner, nonces, MapSet.union(internal, external)}} |
  end
end
```

ADDING PEERS

HANDLE_CAST/2

```elixir
def handle_cast({:newcomer, newcomer}, {owner, _, _} = session) when newcomer == owner do
  {:noreply, session}
end

def handle_cast({:newcomer, newcomer} = payload, {owner, nonces, peers} = session) do
  if MapSet.member?(peers, newcomer) do
    {:noreply, session}
  else
    log("#{newcomer} has joined", "Alert", "👋", :green)

    GenServer.cast(address(newcomer), {:add_peers, peers})
    forward(peers, payload)

    {:noreply, {owner, nonces, MapSet.put(peers, newcomer)}}
  end
end
```

```elixir
def address(room) when is_bitstring(room) do
  room
  |> String.to_atom()
  |> address()
end

def address(room), do: {:chat, room}
```

ADDING PEERS

HANDLE_CAST/2

```
~/D/V/quick_chat  master *$  iex -S mix
.4m < Sun  8 Apr 21:41:15 2018
Erlang/OTP 20 [erts-9.3] [source] [64-bit] [smp:4:4] [ds:4:4:10]
-threads:10] [hipe] [kernel-poll:false] [dtrace]

Interactive Elixir (1.6.4) - press Ctrl+C to exit (type h() ENTER
elp)
iex(1)> {:ok, pid} = GenServer.start(QuickChat.Server, [self()])
🔌 nonode@nohost


{:ok, #PID<0.111.0>}
iex(2)> GenServer.call(pid, :list_peers)
[]
iex(3)> GenServer.cast(pid, {:newcomer, inspect(pid)})
👋 Alert
#PID<0.111.0> has joined

:ok
iex(4)> GenServer.call(pid, :list_peers)
["#PID<0.111.0>"]
```

QUICK CHAT

PRIVATE MESSAGES

PRIVATE MESSAGES

HANDLE_CAST/2

Similar to public message + add peer

```elixir
def handle_cast({:dm, sender, nonce, text}, {_, nonces, _} = session) do
  unless MapSet.member?(nonces, nonce) do
    log(text, sender, "🙈", :blue)
  end

  handle_cast({:newcomer, sender}, session)
end
```

PRIVATE MESSAGES

HANDLE_CALL/3

"Please send a private message"

```elixir
def handle_call({:send_dm, to, text}, {from, _}, {owner, nonces, peers}) when from == owner do
  log(text, "Me (to #{to})", "🙈", :yellow)

  new_nonce = nonce()
  GenServer.cast(address(to), {:dm, me(), new_nonce, text})

  unless MapSet.member?(peers, to) do
    GenServer.cast(address(to), {:add_peers, peers})
    forward(peers, {:newcomer, to})
  end

  {:reply, :ok, {owner, MapSet.put(nonces, new_nonce), MapSet.put(peers, to)}}
end
```

Look familiar?

PRIVATE MESSAGES

HANDLE_CALL/3

```
iex(1)> GenServer.start(QuickChat.Server, [self()], name: :me)
🔌 nonode@nohost


{:ok, #PID<0.120.0>}
iex(2)> GenServer.start(QuickChat.Server, [self()], name: :friend)
🔌 nonode@nohost


{:ok, #PID<0.122.0>}
iex(3)> GenServer.call(:me, {:send_dm, :friend, "Hello over there!"})
🙈 Me (to friend)
Hello over there!

:ok
```

PRIVATE MESSAGES

HANDLE_CALL/3

Just to prove that this works locally

```elixir
def handle_call({:send_dm, to, text}, {from, _}, {owner, nonces, peers}) when from == owner do
  log(text, "Me (to #{to})", "👧", :yellow)

  new_nonce = nonce()
  GenServer.cast(to, {:dm, me(), new_nonce, text})

  unless MapSet.member?(peers, to) do
    GenServer.cast(address(to), {:add_peers, peers})
    forward(peers, {:newcomer, to})
  end

  {:reply, :ok, {owner, MapSet.put(nonces, new_nonce), MapSet.put(peers, to)}}
end

def handle_cast({:dm, sender, nonce, text}, {owner, nonces, peers} = session) do
  log(text, sender, "👧", :blue)

  if MapSet.member?(peers, sender) do
    {:noreply, session}
  else
    GenServer.cast(address(sender), {:add_peers, peers})
    forward(peers, {:newcomer, sender})
    {:noreply, {owner, nonces, MapSet.put(peers, sender)}}
  end
end
```

PRIVATE MESSAGES

HANDLE_CALL/3

```
Interactive Elixir (1.6.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>  GenServer.start(QuickChat.Server, [self()], name: :me)
🔌 nonode@nohost


{:ok, #PID<0.120.0>}
iex(2)> GenServer.start(QuickChat.Server, [self()], name: :friend)
🔌 nonode@nohost


{:ok, #PID<0.122.0>}
iex(3)>  GenServer.call(:me, {:send_dm, :friend, "Hello over there!"})
🙈 Me (to friend)
Hello over there!

🙈 nonode@nohost
Hello over there!

:ok
```

QUICK CHAT

POLISH ✨

POLISH ✨

CLIENT INTERFACE

```elixir
defmodule QuickChat do
  @doc "Set up"
  def start() do GenServer.start(QuickChat.Server, [self()], name: :chat)
end
```

Just for simplicity in demo

```
iex(2)> QuickChat.start()
🔌 nonode@nohost


{:ok, #PID<0.122.0>}
```

POLISH ✨

CLIENT INTERFACE

```elixir
defmodule QuickChat do
  def start(), do: GenServer.start(QuickChat.Server, [self()], name: :chat)

  def peers(), do: GenServer.call(:chat, :list_peers)

  def msg(text), do: GenServer.call(:chat, {:broadcast, text})

  def dm(to, text), do: GenServer.call(:chat, {:send_dm, to, text})
end
```

POLISH ✨

CLIENT INTERFACE

```
~/D/V/quick_chat  master $  iex --name chatroom --cookie monster -
S mix                                    320ms  Sat  7 Apr 22:51:22 201
8
Erlang/OTP 20 [erts-9.3] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async
-threads:10] [hipe] [kernel-poll:false] [dtrace]

Compiling 1 file (.ex)
Interactive Elixir (1.6.4) - press Ctrl+C to exit (type h() ENTER for h
elp)
iex(chatroom@Latte.local)1> import QuickChat
QuickChat
iex(chatroom@Latte.local)2> start
🔌 chatroom@Latte.local


{:ok, #PID<0.127.0>}
iex(chatroom@Latte.local)3> peers
[]
iex(chatroom@Latte.local)4> msg "helo world"
😎 Me
helo world

:ok
iex(chatroom@Latte.local)5> dm("chatroom2@Latte.local", "you there?")
🐶 Me (to chatroom2@Latte.local)
you there?

:ok
iex(chatroom@Latte.local)6> peers
["chatroom2@Latte.local"]
```

```
quick_chat — ~/D/V/quick_chat — beam.smp -- -root /usr/local/Ce
Erlang/OTP 20 [erts-9.3] [source] [64-bit] [smp
-threads:10] [hipe] [kernel-poll:false] [dtrace

Interactive Elixir (1.6.4) - press Ctrl+C to ex
elp)
iex(chatroom2@Latte.local)1> import QuickChat
QuickChat
iex(chatroom2@Latte.local)2> start
🔌 chatroom2@Latte.local


{:ok, #PID<0.118.0>}
iex(chatroom2@Latte.local)3> peers
[]
iex(chatroom2@Latte.local)4> msg "hello world"
😎 Me
hello world

:ok
🐶 chatroom@Latte.local
you there?

iex(chatroom2@Latte.local)5> peers
["chatroom@Latte.local"]
```

QUICK CHAT

NEXT STEPS

- Support “slash commands” (like in Slack)
- Subgroup messaging
- Rooms / channels
- Message history
  - Forwarding to new peers
  - Search history for text
- Allow users to specify a PID
- Tests and type specs