Stateful Processes in Elixir

Julian Doherty

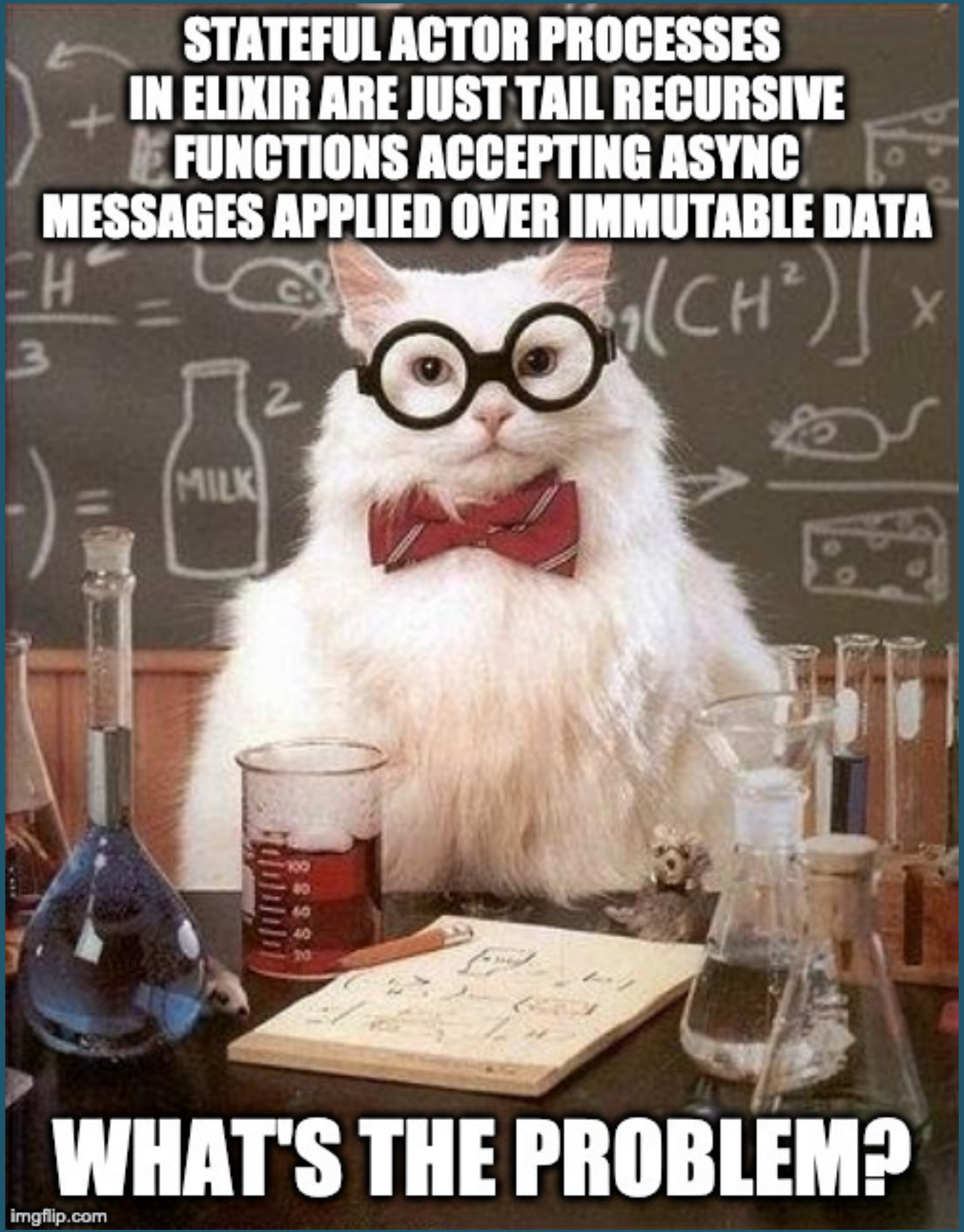
@madlep

```elixir
defmodule MyThing do
  use GenServer
  # ...

  def do_stuff(some_data) do
    # ...
  end
end

{:ok, thing} = MyThing.start_link()
MyThing.do_stuff(thing, "something something")
```

🤔

```
> > "Hey hackerreddit, I need to do a thing in Elixir?"
> "lol Just use GenServer"
```

🤷‍♂️


STATEFUL ACTOR PROCESSES IN ELIXIR ARE JUST TAIL RECURSIVE FUNCTIONS ACCEPTING ASYNC MESSAGES APPLIED OVER IMMUTABLE DATA
WHAT'S THE PROBLEM?
imgflip.com

I can't blindly follow a recipe book.

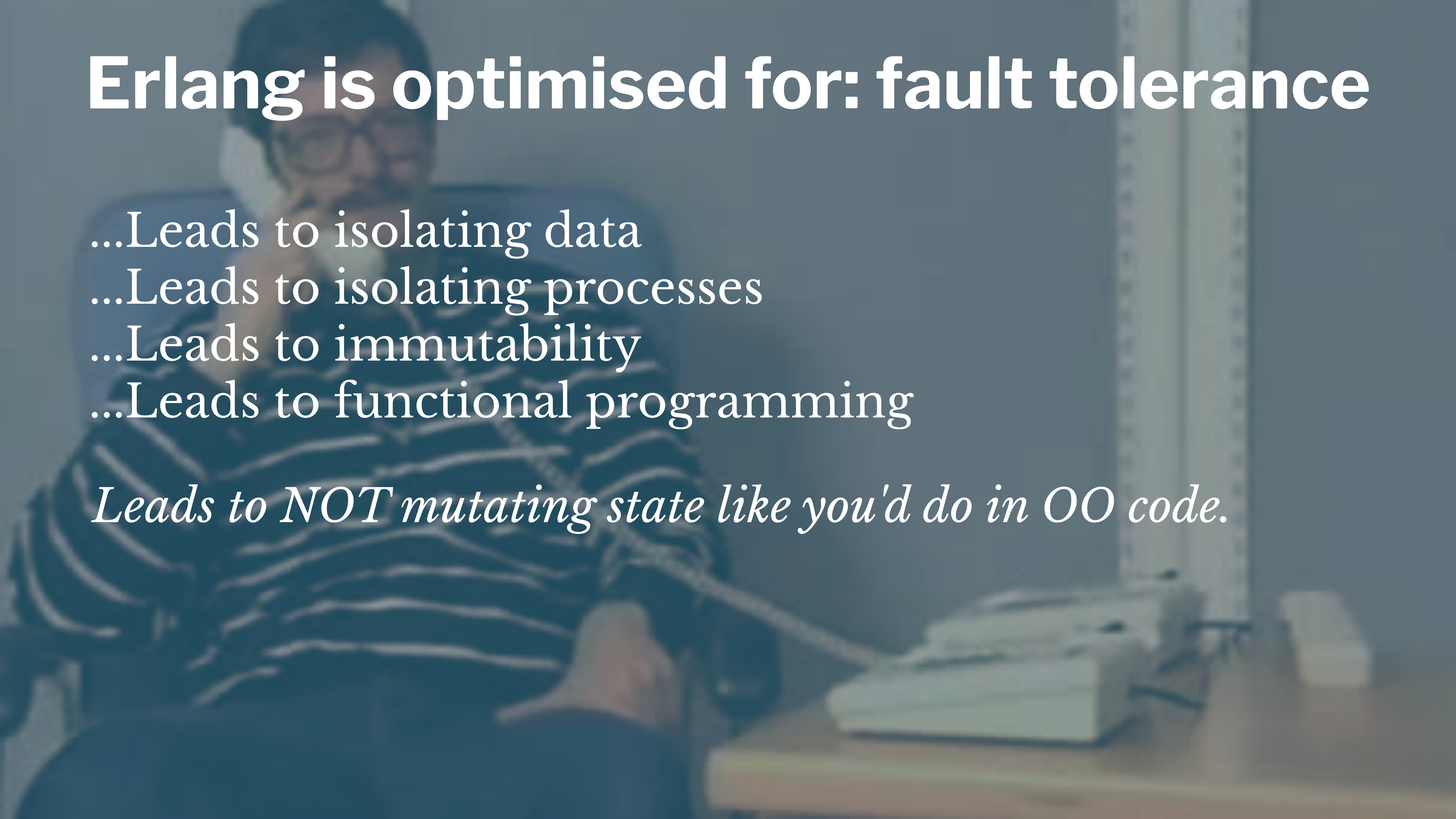
I need to "get" how it works under the hood

The one goal for today:

"how stateful processes work" -> your head

Enough for you to reason about Elixir code you see in the wild

Erlang is optimised for: fault tolerance



Erlang is optimised for: fault tolerance

...Leads to isolating data
...Leads to isolating processes
...Leads to immutability
...Leads to functional programming

Leads to NOT mutating state like you'd do in OO code.

What is "state"?

Your data
Stuff that changes
What you need to manipulate to do useful work

What is "stateful"?

Keeping data in memory somewhere.
If you've got a reference to it, you can "do stuff".

If you're doing OO, you're doing "stateful" (probably)

What is "stateless"?

Not keeping data in memory
Just passing it from function to function
Transforming it along the way

If you're doing FP, you're doing "stateless" (probably)

So if we can't have state, and we can't mutate state?...

Functions!

"Do some work, then call yourself with the changed state to do more work, repeat until done (or forever)"

Functional shopping cart

```elixir
defmodule ShoppingCart do
  def init() do
    []
  end

  def add_item(cart, item) do
    [item | cart]
  end
end

cart = ShoppingCart.init
cart2 = ShoppingCart.add_item(cart, "milk")
cart3 = ShoppingCart.add_item(cart2, "bread")

IO.inspect cart3
# ["bread", "milk"]
```

Functional shopping cart

cool... but:
- no way to share state between processes
- stuck in single process land
- **not fault tolerant.** If it crashes, the process it's in crashes

Need a way to do shared state…
While not having shared state
(safely)

So if we can't have shared state?...

Tail Recursive Functions!
And...
Processes!

Tail Recursive Functions

```
defmodule ShoppingCart do
  # ...

  def count_items(cart, count \\ 0)

  defp count_items([], count), do: count

  defp count_items([_item|cart], count), do: count_items(cart, count + 1)
end

# ...
IO.inspect ShoppingCart.count_items(cart3)
# 2
```

Processes

```elixir
defmodule ShoppingCart do
  def start(), do: spawn(fn -> loop([]) end)

  def loop(cart) do
    receive do
      {:add_item, item} ->
        [item | cart] |> loop()
      {:count_items, from} ->
        send(from, {:count_response, count_items(cart, 0)})
        loop(cart)
    end
  end
  # ...
end

cart = ShoppingCart.start()
send(cart, {:add_item, "milk"})
send(cart, {:add_item, "bread"})
send(cart, {:count_items, self()})
receive do
  {:count_response, count} -> IO.inspect(count)
end
```

Cool... but a lot of boilerplate

Let's extract some abstractions around process plumbing

Generic server for stateful processes

```
defmodule MyServer do
  def start(mod), do: spawn(fn -> apply(mod, :init, []) |> loop(mod) end)

  def call(server, args) do
    send(server, {:call, args, self()})
    receive do
      {:call_response, result} -> result
    end
  end

  def cast(server, args) do
    send(server, {:cast, args})
  end

  defp loop(state, mod) do
    receive do
      {:cast, args} ->
        apply(mod, :handle_cast, [args, state]) |> loop(mod)
      {:call, args, from} ->
        result = apply(mod, :handle_call, [args, state])
        send(from, {:call_response, result})
        loop(state, mod)
    end
  end
end
```

And server implementation for our cart

```elixir
defmodule ShoppingCart do
  def init() do
    []
  end

  def handle_cast({:add_item, item}, cart), do: [item | cart]
  def handle_call(:count_items, cart), do: count_items(cart, 0)

  # ...
end

cart = MyServer.start(ShoppingCart)
MyServer.cast(cart, {:add_item, "milk"})
MyServer.cast(cart, {:add_item, "bread"})
IO.inspect MyServer.call(cart, :count_items)
```

This is GenServer!

That's 90% of what `use GenServer` does for you

Let's use `GenServer` then

```elixir
defmodule ShoppingCart do
  use GenServer

  def init(_args) do
    {:ok, []}
  end

  def handle_cast({:add_item, item}, cart) do
    {:noreply, add_item(cart, item)}
  end

  def handle_call(:count_items, _from, cart) do
    {:reply, count_items(cart), cart}
  end

  # ...
end

{:ok, cart} = GenServer.start(ShoppingCart, [])
GenServer.cast(cart, {:add_item, "milk"})
GenServer.cast(cart, {:add_item, "bread"})
IO.inspect GenServer.call(cart, :count_items)
```

Convention is to provide nicer client API

```
defmodule ShoppingCart do
  use GenServer

  def start(_args), do: GenServer.start(ShoppingCart, [])

  def add_item(cart, item), do: GenServer.cast(cart, {:add_item, item})

  def count_items(cart), do: GenServer.call(cart, :count_items)

  # ...
end

{:ok, cart} = ShoppingCart.start([])
ShoppingCart.add_item(cart, "milk")
ShoppingCart.add_item(cart, "bread")
IO.inspect ShoppingCart.count_items(cart)
```

We've lost some things though

- GenServer plumbing is mixed up with application logic
- harder to test in isolation
- harder to understand and reason about

API/Server/Impl pattern

split up responsibilities
- **API** (or "base") module is called from outside, nice interface. Does GenServer calls/casts
- **Server** module implements GenServer behaviour, delegates to...
- **Impl** module does the actual business logic and manages state

Splitting APIs, Servers, and Implementations in Elixir
https://pragdave.me/blog/2017/07/13/decoupling-interface-and-implementation-in-elixir.html

```elixir
defmodule ShoppingCart do
  def start(_args) do
    GenServer.start(ShoppingCart.Server, [])
  end

  def add_item(cart, item) do
    GenServer.cast(cart, {:add_item, item})
  end

  def count_items(cart) do
    GenServer.call(cart, :count_items)
  end
end
```

```elixir
defmodule ShoppingCart.Server do
  use GenServer

  def init(_args), do: {:ok, ShoppingCart.Impl.create()}

  def handle_cast({:add_item, item}, cart) do
    {:noreply, ShoppingCart.Impl.add_item(cart, item)}
  end

  def handle_call(:count_items, _from, cart) do
    {:reply, ShoppingCart.Impl.count_items(cart), cart}
  end
end
```

```elixir
defmodule ShoppingCart.Impl do
  def create(), do: []

  def add_item(cart, item), do: [item | cart]

  def count_items(cart), do: count_items(cart, 0)

  defp count_items([], count) do
    count
  end

  defp count_items([_item|cart], count) do
    count_items(cart, count + 1)
  end
end

{:ok, cart} = ShoppingCart.start([])
ShoppingCart.add_item(cart, "milk")
ShoppingCart.add_item(cart, "bread")
IO.inspect ShoppingCart.count_items(cart)
```

What did all that buy us?

LET IT CRASH
memeg

What did all that buy us?

- The shopping cart is now isolated and fault tolerant.
- Our app can now scale across multi core
- We can supervise or app, and set different restart policies if they fail

This is *everywhere* in Elixir

- Agent
- Task
- GenStage
- Flow
- LiveView
- Scenic
- Supervisors
- More...

If you remember one thing:

When you see

```
use GenServer
```

Mentally picture code running as a separate, isolated process.

There is a function, that calls itself in a loop...
sitting there waiting to receive your messages...
and send messages back...
asynchronously...

Thank you!

Questions?