

# Coupling Data and Behaviour

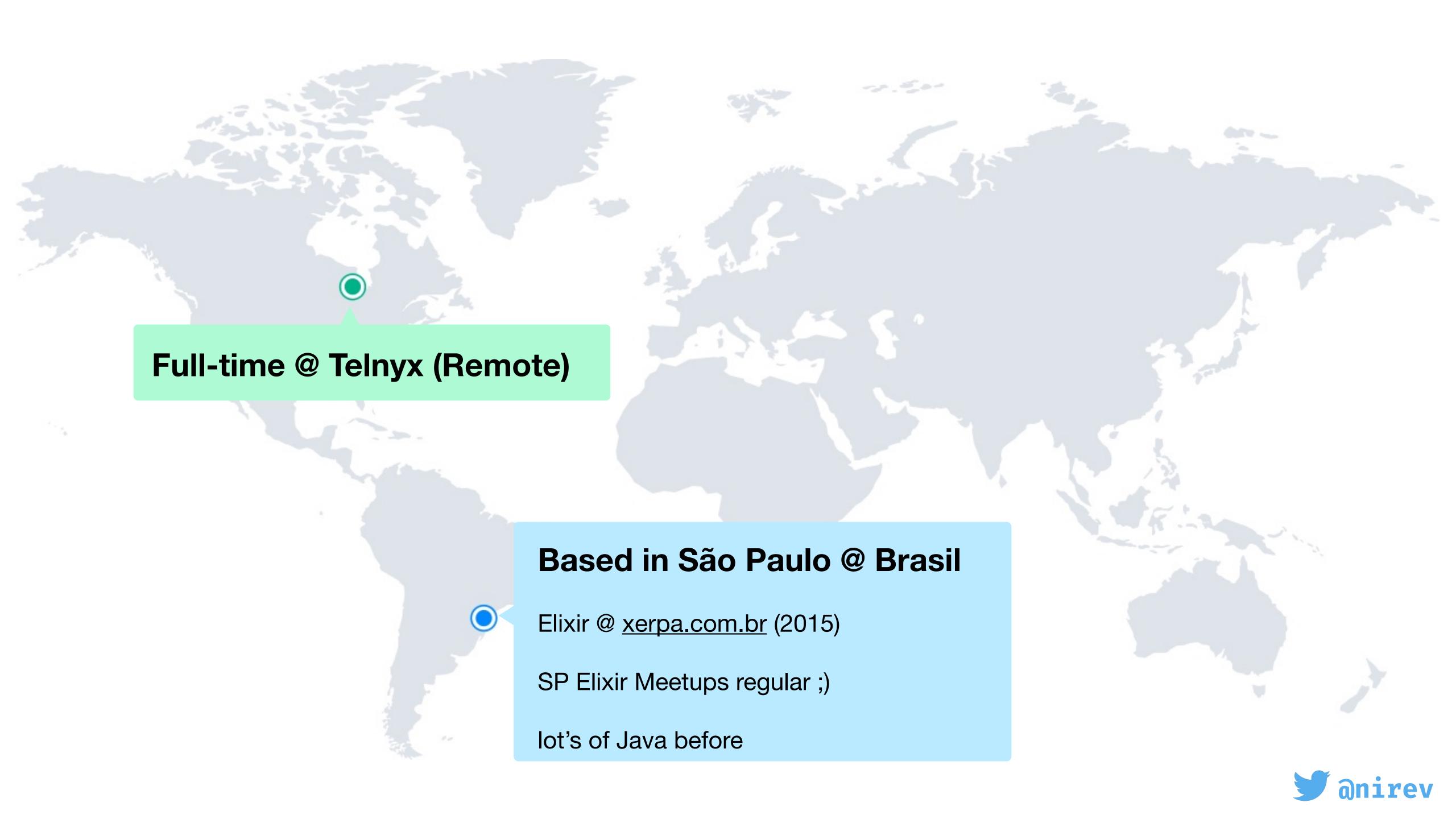
(in Elixir)



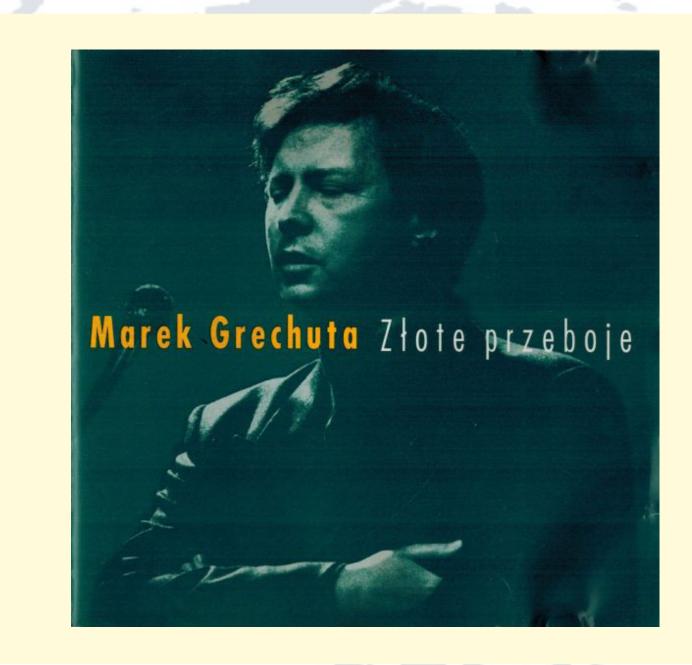
2018-04-17 Guilherme de Maio











#### Based in São Paulo @ Brasil

Elixir @ <u>xerpa.com.br</u> (2015)

SP Elixir Meetups regular;)

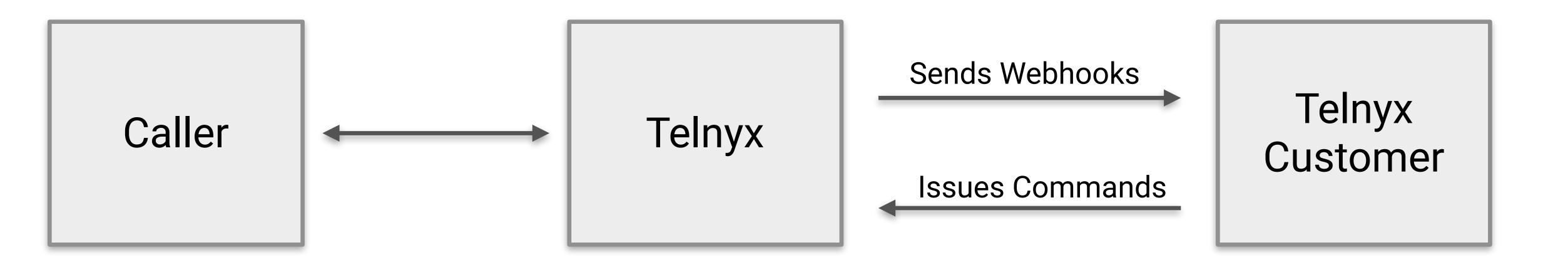
lot's of Java before



# TELNYX



# TELNYX





### Data and Behaviour

How so?



One of the **fundamental principles** of object-oriented design is **to combine data and behavior**, so that the basic elements of our system (objects) combine both together.

Michael Fowler



Finally, another downside to object-oriented programming is the tight coupling between function and data. In fact, the Java programming language forces you to build programs entirely from class hierarchies, restricting all functionality to containing methods in a highly restrictive "Kingdom of Nouns" (Yegge 2006).

Fogus/Houser | The Joy of Clojure



### Call Control

Binary strings via socket



#### Call Control: needs

- Several commands sent via socket as binary strings
- Each command can take different parameters
- Adding new commands should be easy
- Sending commands should be the same for all
- Crossconcerns: logging, authorization, etc...





Beware: pseudocode



```
interface Command {
  String render();
}
```





```
class Commander {
   def send_command(Command command) {
     command_str = command.render()
     Socket.send(command_str)
   }
}
```





```
class Play impl Command {
  string uuid;
  string audio;
  string channels;
  def render() {
    "playback #{uuid} " +
      "#{audio} #{channels}"
```





```
class Play impl Command {
  string uuid;
  string audio;
  string channels;
  def render() {
    "playback #{uuid} " +
      "#{audio} #{channels}"
```

```
class Record impl Command {
  string uuid;
  string output;
  string format;
  string channels
 def render() {
    "record #{uuid} " +
      "#{output} " +
      "format=#{format}" +
      "channels=#{channels}
```





- Adding new commands should be easy

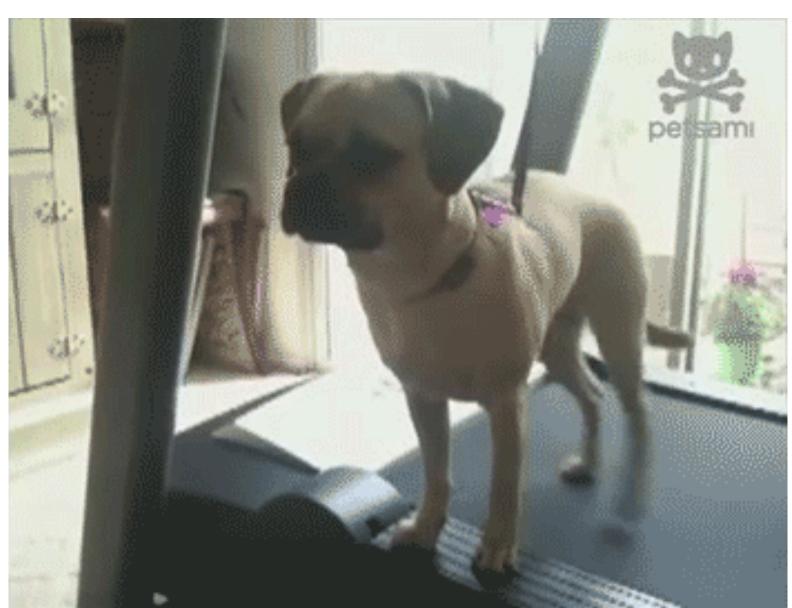
```
class Whatever impl Command {
   string uuid;
   string audio;
   string channels;

def render() {...}
}
```





```
class Commander {
  def send_command(command) {
    command_str = command.render()
    Socket.send(command_str)
}
```



#### Properties:

- Extensible
- Open/Closed





```
class Whatever impl Command {
  string uuid;
  string audio;
  string channels;

def render() {...}
}
```

What if we need more methods??





```
class Whatever impl Command {
  string uuid;
  string audio;
  string channels;
                                What if we need more methods??
  def render() {...}
                       interface Command {
 def log() {...}
                         String render();
                         void log();
```









- Several commands sent via socket as binary strings 🗹
- Each command can take different parameters <a>V</a>
- Adding new commands should be easy
- Sending commands should be the same for all
- Crossconcerns: logging, authorization, etc... 🔽







## Functional Commands;)



```
defmodule Commander do
  def send(command) do
    command ▷ render() ▷ socket_send()
  end
end
```





```
defmodule Play do
  defstruct [
    :uuid,
    :audio,
    :channels]
end
```

```
defmodule Record do
  defstruct [
   :uuid,
   :output,
   :format,
   :channels]
end
```





```
defmodule Commander do
  defmodule Play do
                         def send(command) do
    defstruct [
                           command > render() > socket_send()
      :uuid,
                         end
      :audio,
      :channels]
                         def render(%Play{}) = p) do
  end
                           "playback 辉p.uuid} 辉p.audio} 辉p.channels}"
                         end
  defmodule Record do
    defstruct [
                         def render(%Record{} = r) do
    :uuid,
                           "record #{r.uuid} " ◇ "#{r.output} " ◇
    :output,
                            "format=#{r.format}" ♦ "channels=#{r.channels}
    :format,
                         end
    :channels]
                       end
A end
                                                                      Onirev
```

So far:

How to add a new command?

How to add a new function?





So far:

How to add a new command?

How to add a new function?







This is the way of doing this with Erlang.

With pattern-matching it's possible to dispatch per struct but it's not possible to do it following "open/closed" principle.

This is the sort of thing that makes the "raison d'être" of Elixir.





The problem I have with Erlang is that the language is somehow too simple, making it very hard to eliminate boilerplate and structural duplication. Conversely, the resulting code gets a bit messy, being harder to write, analyze, and modify. After coding in Erlang for some time, I thought that functional programming is inferior to 00, when it comes to efficient code organization.

Sasa Juric | Why Elixir



# Protocols:)



```
defprotocol Command do
  def render(cmd)
end
```





A end

```
defimpl Command, for: Play do
defmodule Play do
                      def render(p) do
  defstruct [
                        "playback #{p.uuid} #{p.audio} #{p.channels}"
    :uuid,
                      end
    :audio,
                    end
    :channels]
end
                    defimpl Command, for: Record do
                      def render(r) do
defmodule Record do
                        "record #{r.uuid} #{r.output} " <>
  defstruct [
                         "format=#{r.format} channels=#{r.channels}"
  :uuid,
                      end
  :output,
                    end
  :format,
  :channels]
```

Onirev





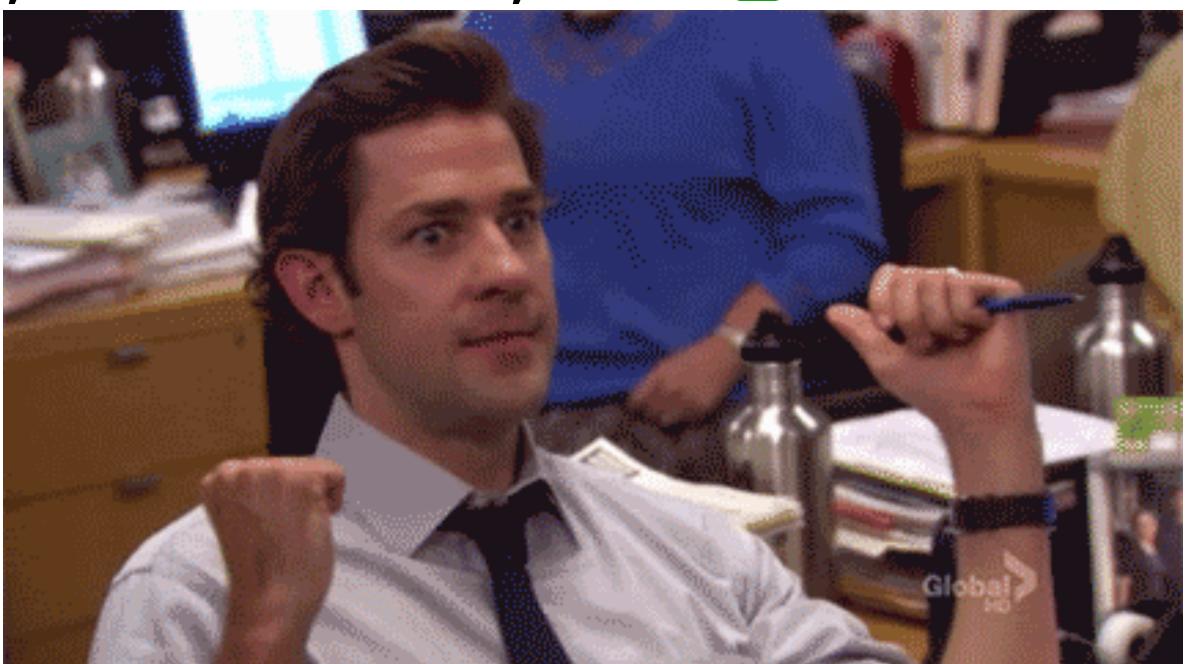
- Several commands sent via socket as binary strings 🔽
- Each command can take different parameters <a>V</a>
- Adding new commands should be easy 🔽
- Sending commands should be the same for all 💟
- Crossconcerns: logging, authorization, etc... 🔽





#### Protocols

- Several commands sent via socket as binary strings 🗹
- Each command can take different parameters <a>V</a>
- Adding new commands should be easy 🔽
- Sending commands should be the same for all
- Crossconcerns: logging, authorization, etc... 🔽







# Protocols: what, why, how



## WHAT



## Protocols: inspired by Clojure



- Provide a high-performance, dynamic polymorphism construct as an alternative to interfaces
- Support best part of interfaces, while avoiding the drawbacks





## Protocols: inspired by Clojure

```
(defprotocol P
  (foo [x])
  (bar [x y]))
(deftype Foo [a b c]
  (foo [x] a)
  (bar [x y] (+ c y))
(foo (Foo. 1 2 3))
\Rightarrow 1
(bar (Foo. 1 2 3) 42)
\Rightarrow 45
```





## Protocols: inspired by Clojure

```
(defprotocol P
  (foo [x])
  (bar [x y]))
(deftype Foo [a b c]
  (foo [x] a)
  (bar [x y] (+ c y)))
(foo (Foo. 1 2 3))
(bar (Foo. 1 2 3) 42)
\Rightarrow 45
```

```
defprotocol P do
  def foo(x)
  def bar(x, y)
end
defmodule Foo do
  defstruct [:a, :b, :c]
  defimpl P do
    def foo(f), do: f.a
    def bar(f, y) do: f.c + y
  end
end
```





```
defprotocol Size do
  def size(data)
end
```





```
-> Implement it for native types
defprotocol Size do
  def size(data)
end
defimpl Size, for: BitString do
  def size(string), do: byte_size(string)
end
defimpl Size, for: Map do
  def size(map), do: map_size(map)
end
```





```
-> Implement it for native types
defprotocol Size do
  def size(data)
end
defimpl Size, for: BitString do
                                              iex> Size.size "abacate"
  def size(string), do: byte_size(string)
end
defimpl Size, for: Map do
                                              iex> Size.size %{a: 1, b: 2}
  def size(map), do: map_size(map)
end
```





```
-> Implement it for structs
defprotocol Size do
  def size(data)
end
defmodule Bag do
                                   iex> Size.size %Bag{items: [1,2,3]}
  defstruct [:items]
  defimpl Size do
    def size(bag), do: Enum.count(bag.items)
  end
end
```





-> Implement it for structs, decoupled from module definition
defprotocol Size do
 def size(data)

```
defmodule Bag do
  defstruct [:items]
end
```

```
defimpl Size, for Bag do
  def size(bag), do: Enum.count(bag.items)
end
```



end



```
-> Have defaults
defprotocol Size do
  def size(data)
end
defimpl Size, for: Any do
  def size(_), do: 0
end
defmodule Sizeless do
                            iex> Size.size %Sizeless{wat: :wat}
  @derive [Size]
  defstruct [:wat]
end
```



```
-> Have defaults
   defprotocol Size do
     def size(data)
   end
   defimpl Size, for: Any do
     def size(%{size: size}), do: size
     def size(_), do: 0
   end
   defmodule Sizeable do
                              iex> Size.size %Sizeable{size: 42}
     @derive [Size]
                               42
     defstruct [:size]
_ end
```

Onirev







# HOW



```
defmodule Data do
  defstruct [:x]
end

%Data{x: 1} = %{__struct__: Data, x: 1}
```





```
defmodule Protocolz do
  def dispatch(function_name, data) do
    struct_module = data.__struct__
    :erlang.apply(struct_module, function_name, [data])
  end
end
```





```
defmodule Play do
    defstruct [:id, :audio]
    def render(p) do
        "playback #{p.id} #{p.audio}"
    end
end
defmodule Record do
    defstruct [:id, :output]
    def render(r) do
        "record #{r.id} #{r.output}"
    end
end
```

```
play = %Play{id: "19831", audio: "audio.mp3"}
Protocolz.dispatch(:render, play)

record = %Record{id: "908301931", output: "record.mp3"}
Protocolz.dispatch(:render, record)
```





```
defmodule Protocolz do
 defmacro defimplz(protocol, [for: struct_module], [do: block]) do
    quote do
      defmodule Module.concat([unquote(protocol), unquote(struct_module)]) do
        unquote(block)
      end
    end
  end
  def dispatch(protocol, function_name, data) do
    struct_module = data.__struct__
    impl_module = Module.concat(protocol, struct_module)
    :erlang.apply(impl_module, function_name, [data])
  end
end
                                                                      Onirev
```



```
defimplz Command, for: Play do
  def render(p) do
   "playback #{p.id} #{p.audio}"
  end
end
```

```
defimplz Command, for: Record do
  def render(r) do
    "record #{r.id} #{r.output}"
  end
end
```

```
play = %Play{id: "19831", audio: "audio.mp3"}
  Protocolz.dispatch(Command, :render, play) # Command.Play.render()

record = %Record{id: "908301931", output: "record.mp3"}
  Protocolz.dispatch(Command, :render, record) # Command.Record.render()
```





```
defmacro defprotocol(name, do: block) do
   quote do
    defmodule unquote(name) do
       import Protocol, only: [def: 1]
    # Invoke the user given block
    _ = unquote(block)
   end
   end
end
```





```
defmacro def({name, _, args}) do
  quote do
    name = unquote(name)
    arity = unquote(arity)
    Kernel.def unquote(name)(unquote_splicing(args)) do
      impl_for!(term).unquote(name)(unquote_splicing(args))
    end
  end
end
```



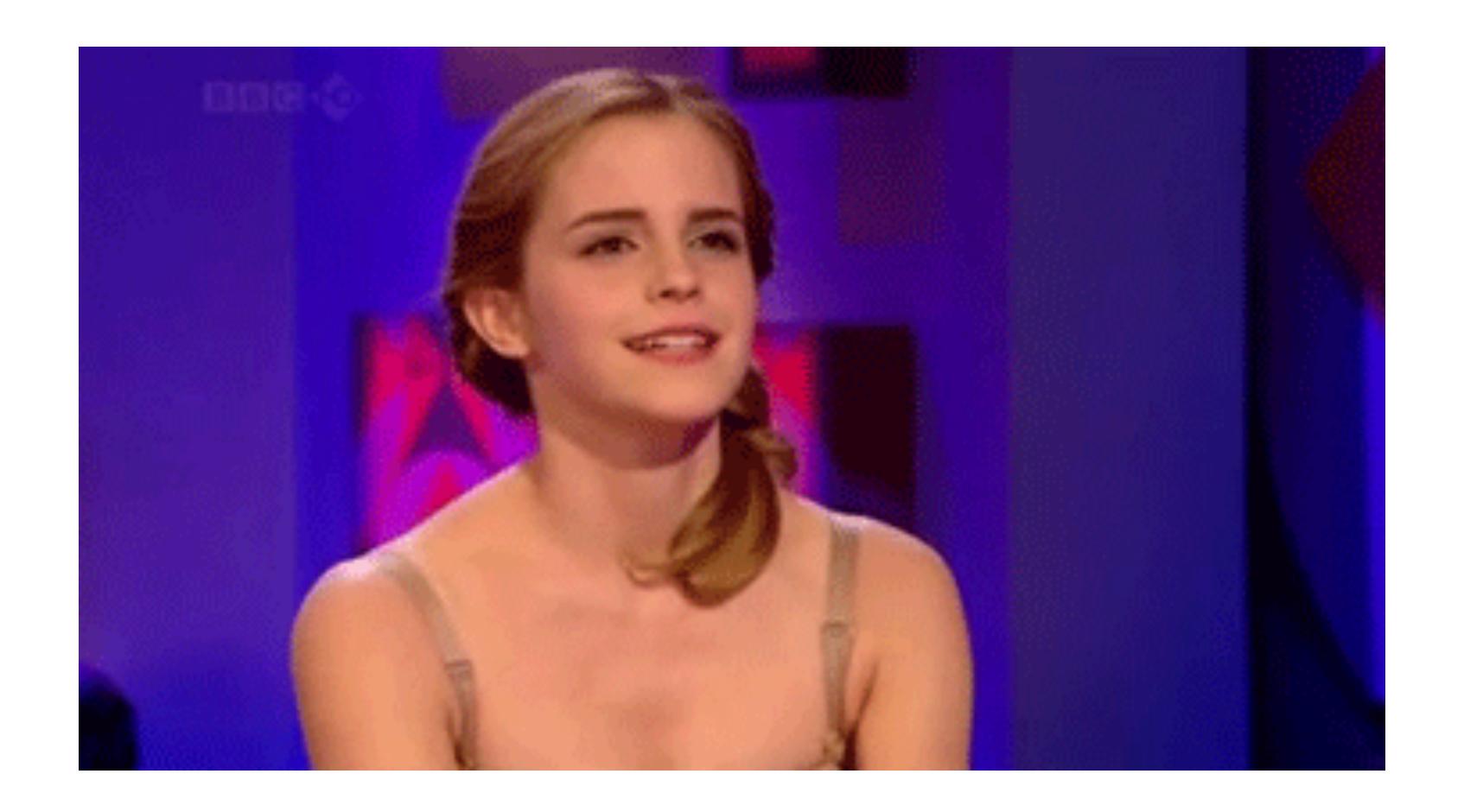


```
defprotocol Command do
  def render(x)
end
```

```
defmodule Command do
  def render(x) do
    impl_for!(x).render(x)
  end
  defp impl_for!(struct) do
    target = Module.concat(__MODULE__, struct)
    case Code.ensure_compiled?(target) do
      true → target.__impl__(:target)
      false → nil
    end
  end
end
```











```
defmodule Command do
  def render(x) do
    impl_for!(x).render(x)
  end
  defp impl_for!(struct) do
    target = Module.concat(__MODULE__, struct)
    case Code.ensure_compiled?(target) do
      true → target.__impl__(:target)
      false → nil
    end
  end
end
```





```
defmodule Command do
  def render(x) do
    impl_for!(x).render(x)
  end
 defp impl_for!(struct)

target = Module.
MODULE__, struct)
                      _compiled?(target) do
                 __get.__impl__(:target)
  end
end
```





## WHY



## String.chars

```
defmodule Product do
  defstruct title: "", price: 0

defimpl String.Chars do
   def to_string(%Product{title: title, price: price}) do
    "#{title}, $#{price}"
   end
  end
end
```





#### Poison

```
defmodule Person do
 @derive [Poison.Encoder]
 defstruct [:name, :age]
end
defimpl Poison. Encoder, for: Person do
  def encode(%{name: name, age: age}, opts) do
    Poison.Encoder.BitString.encode("#{name} (#{age})", opts)
  end
end
```





#### infinitered/elasticsearch-elixir

```
defimpl Elasticsearch.Document, for: MyApp.Post do
  def id(post), do: post.id
  def type(_post), do: "post"
  def parent(_post), do: false
  def encode(post) do
    %{
      title: post.title,
      author: post.author
  end
```





# What about Behaviour?



#### Behaviour vs Protocols

```
defprotocol Command do
    @spec render(t) :: String.t
    def render(cmd)
end

defmodule Commander do
    @type r :: :ok | :error
    @callback render(Command.t) :: r
end
```





#### Behaviour vs Protocols

```
defprotocol Command do
 @spec render(t) :: String.t
 def render(cmd)
end
defmodule Commander do
  atype r :: :ok | :error
  @callback send(Command.t) :: r
end
```

```
defmodule FakeCommander do
 abehaviour Commander
 def send(_command), do: :ok
end
defmodule SocketCommander do
  abehaviour Commander
  def send(command) do
  end
end
```





Protocols	Behaviours
Dispatch on type	Dispatch on module
Contract for Data	Contract for modules
Elixir-only	Erlang too
Ex: Encoding/ Serializing structs	Ex: sending email via SMTP/IMAP, API or mocks



# Bottomline?



Coupling data and behaviour can be GOOD, and Elixir protocols are f!%@#%@ awesome!







Guilherme de Maio guilherme@taming-chaos.com

