

# Thinking Functionally

Elixir Taiwan Meetup  
May 25, 2016

Jake Morrison [jake@cogini.com](mailto:jake@cogini.com)



# What is Functional Programming?

- More sophisticated type systems, e.g. Haskell
- No side effects: function outputs depend only on inputs
  - No shared state
  - Concurrency
- Pattern matching



# Benefits

- Easier to test
- Easier to deal with faults in production
- All state is in function parameters, so logs are good
- Message passing



# Functional vs Object Oriented: Types

- OO connects behavior with types, i.e. object methods
- Functional programming uses types for safety
- Modern functional programming languages use type inference to reduce programmer overhead
- "If it will compile, it's correct"



# Erlang types

- Pattern matching at runtime
- “Let it crash”
- Supervisors retrying
- Hot code updates



# Type checking

- Optional type checking
- Typespecs
- Dialyzer
- Tagged tuples
  - `{:ok, value}` vs `{:error, reason}`



# Elixir types

- Structs are simply wrappers on Maps
- ```
defmodule User do
  defstruct name: "John", age: 27
end
```
- ```
iex> %User{}
%User{age: 27, name: "John"}
iex> %User{name: "Meg"}
%User{age: 27, name: "Meg"}
```
- ```
iex> is_map(john)
true
iex> john.__struct__
User
```



# Immutable Data

- It's a good thing
- Erlang does not allow mutating variables
  - Actually re-binding
- Elixir allows it, but it goes against the core of the language
- If you are mutating variables, you are probably doing something wrong
- "Help Doctor, my variables are not varying!"





# Functional vs Object Oriented: Nouns vs Verbs

- OO: No unbound methods
- FP: Standard algorithms with "meta-programming", lambda functions
- Lambda functions, Ruby "blocks" becoming popular
- Execution in the Kingdom of Nouns:  
<http://steve-yegge.blogspot.tw/2006/03/execution-in-kingdom-of-nouns.html>



# Functional vs Object Oriented: Polymorphism

- OO: Inheritance
- FP: Protocols



# Protocols

```
- defprotocol Blank do
  @doc "Returns true if data is considered blank/empty"

  def blank?(data)
  end

end
```



# Protocols

- ```
defimpl Blank, for: Integer do
  def blank?(_), do: false
end
```
- ```
defimpl Blank, for: List do
  def blank?([]), do: true
  def blank?(_), do: false
end
```
- ```
defimpl Blank, for: Map do
  # Keep in mind we could not pattern match on %{} because
  # it matches on all maps. We can however check if the size
  # is zero (and size is a fast operation).
  def blank?(map), do: map_size(map) == 0
end
```



# Protocols

- defimpl Blank, for: Atom do  
  def blank?(false), do: true  
  def blank?(nil), do: true  
  def blank?( \_ ), do: false  
end
- defimpl Blank, for: User do  
  def blank?(\_), do: false  
end



# Protocols: JSON

- `iex> IO.puts Poison.Encoder.encode([1, 2, 3], [])`  
`"[1,2,3]"`
- `defimpl Poison.Encoder, for: Person do`  
  `def encode(%{name: name, age: age}, options) do`  
    `Poison.Encoder.BitString.encode("#{name} (#{age})",`  
  `options)`  
  `end`  
`end`



# Higher Order Programming

- Using functions to “specialize” common algorithms
- Recursion is nice, but generally not the right solution
- Standard algorithms make intent clearer
- Fewer bugs



# Higher Order Programming: List Comprehensions

- `iex> IO.puts Poison.Encoder.encode([1, 2, 3], [])`  
`"[1,2,3]"`
- `defimpl Poison.Encoder, for: Person do`  
  `def encode(%{name: name, age: age}, options) do`  
    `Poison.Encoder.BitString.encode("#{name} (#{age})",`  
  `options)`  
  `end`  
`end`





# Higher Order Programming: Map

- `iex> Enum.map([1, 2, 3], fn x -> x 2 end)`  
`[2, 4, 6]`
- `iex> Enum.map(%{1 => 2, 3 => 4}, fn {k, v} -> k v end)`  
`[2, 12]`



# Higher Order Programming: Fizz Buzz

```
- defmodule FizzBuzz do
  def fizzbuzz_check(n) do
    case {rem(n, 3), rem(n, 5)} do
      {0, 0} -> "FizzBuzz"
      {0, _} -> "Fizz"
      {_, 0} -> "Buzz"
      {_, _} -> n
    end
  end
end

def fizzbuzz do
  IO.inspect Enum.map(1..100, fizzbuzz_check/1)
end
end
```



# Higher Order Programming: Fold / Reduce

- iex> List.foldl([1, 2, 3], 0, fn (x, acc) -> x + acc end)  
6



# Higher Order Programming: Streams

```
◆ nums = Stream.iterate(1, &(&1 + 1))
  fizz = Stream.cycle [ "", "", "Fizz" ]
  buzz = Stream.cycle [ "", "", "", "", "Buzz" ]
  fizzbuzz = Stream.zip(fizz, buzz)
  |> Stream.zip(nums)
  |> Stream.map(fn
    {{"", ""}, number} -> number
    {{fizzword, buzzword}, _number} -> fizzword <> buzzword
  end)
  fizzbuzz |> Stream.take(30) |> Enum.join("\n") |> IO.puts
```

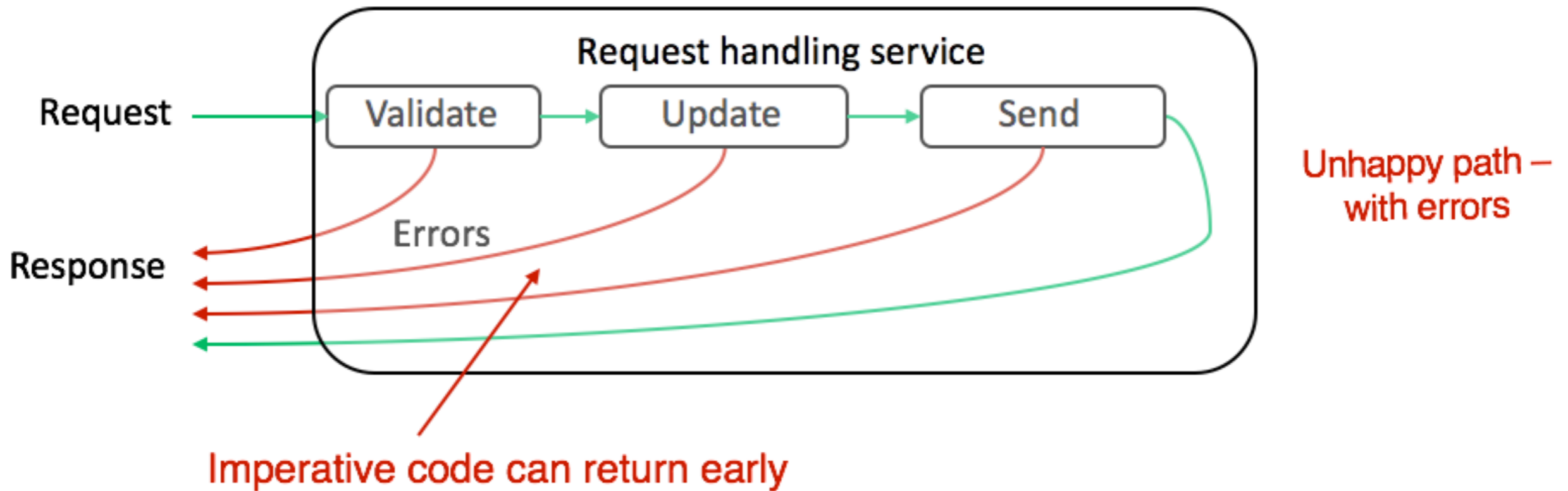


# Phoenix Request Processing

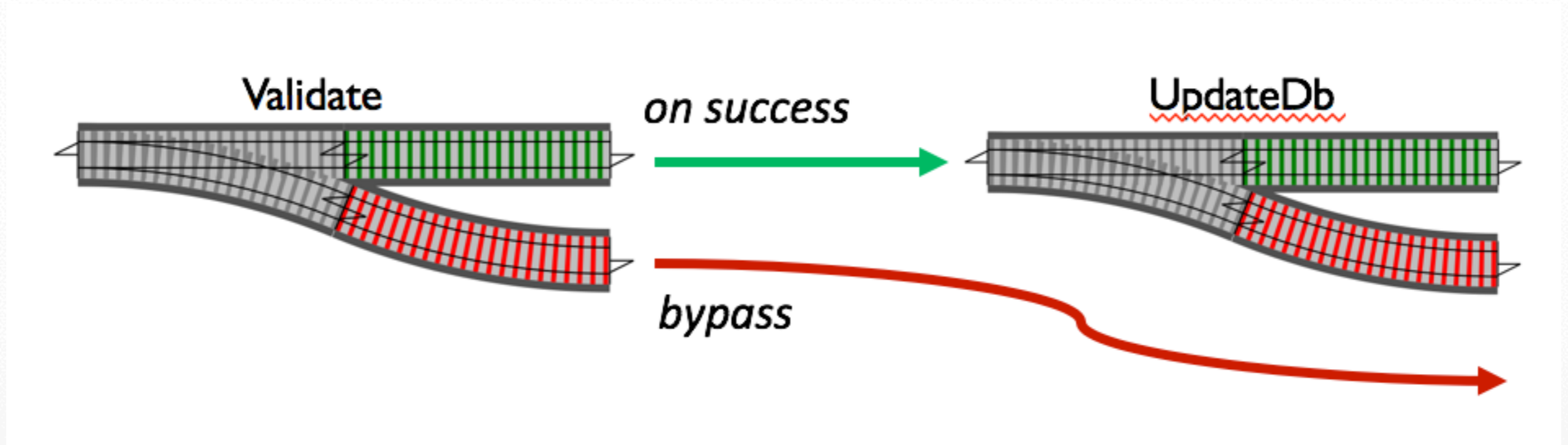
- Handling a request is just a series of transformations
- Take a request as input, transform it into a response
- Plug framework



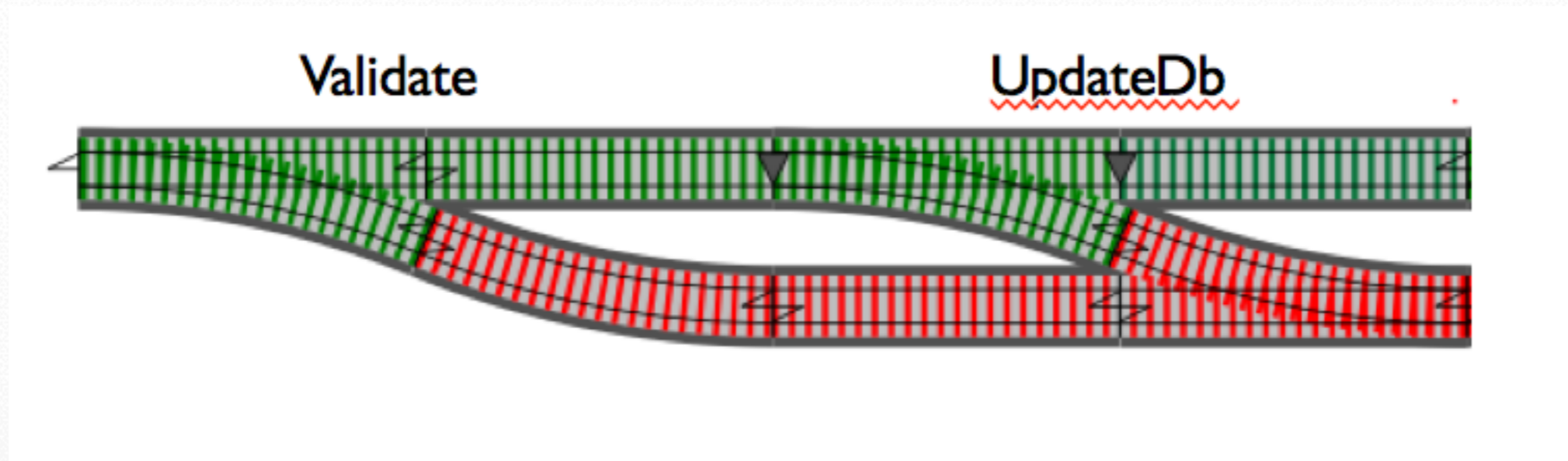
# Railway Oriented Programming



# Railway Oriented Programming

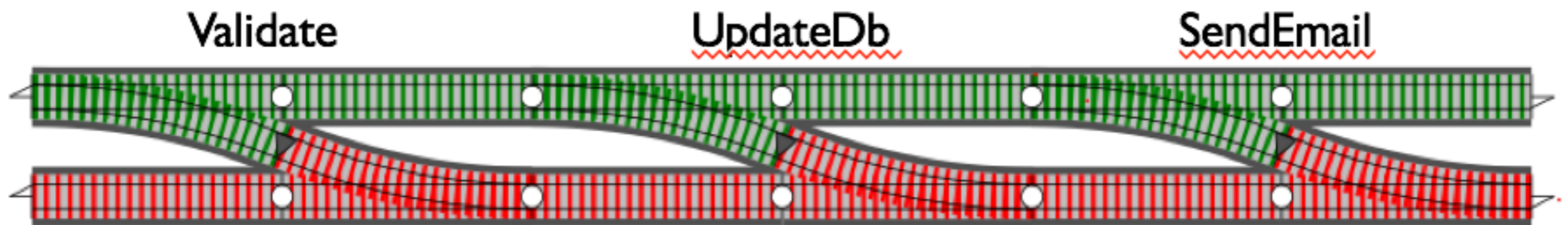


# Railway Oriented Programming





# Railway Oriented Programming



- <http://zohaib.me/railway-programming-pattern-in-elixir/>



# Heresy

- Small functions instead of testing
- Relational model is primary, not ORM
- Erlang is a \*truly\* object oriented language, unlike these pretenders



# Questions?

