Functional thinking with Elixir

Alexei Sholik

About me

- Programming language nerd
- Elixir user and contributor since 2012
- Elixir/iOS developer @ PSPDFKit
- github.com/alco
- @true_droid

What is Elixir?

Dynamic, **functional**, **concurrent** language for building **fault-tolerant**, **scalable**, **maintainable** systems

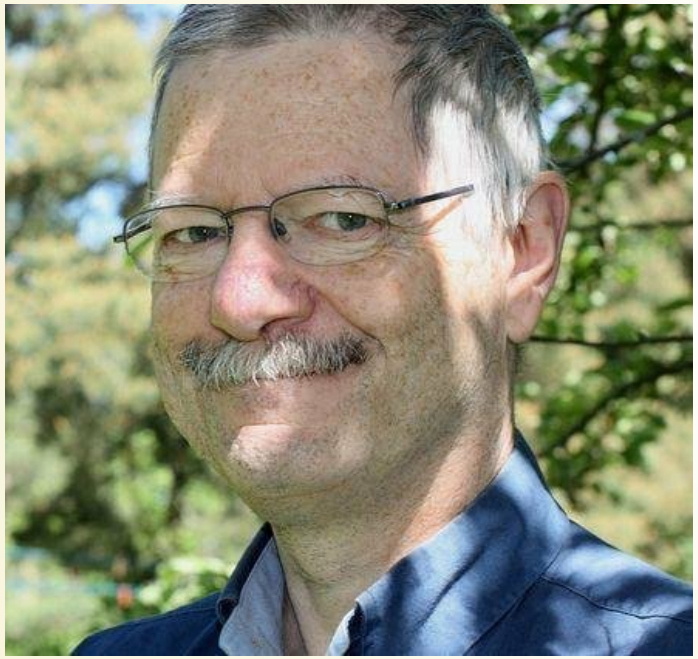
Brief history 1/4

- Created by Ericsson to program telephone switches
- First proprietary release in 1986
- Open source release in 1998

Brief history 2/4

- Around 2011, José Valim considers "thread-safe Rails" an impossibility
- Starts work on Elixir in 2012

Brief history 3/4



One year later, Joe Armstrong likes it

Brief history 4/4

- July 2014: first ElixirConf in Austin, TX
- September 2014: Elixir 1.0.0 released
- Today: active community, use in production, bright future

Elixir is...


love is...
YOUR TOUCH A SMILE
BEING THERE
HELPING
GIVING
A KIND WORD
A GLOW
STEADFAST
SHARING
TENDER
... made up of so many
little things.

Elixir is dynamic

- Dynamic type system
- Interpreted byte code
- Allows to change program behavior at run time

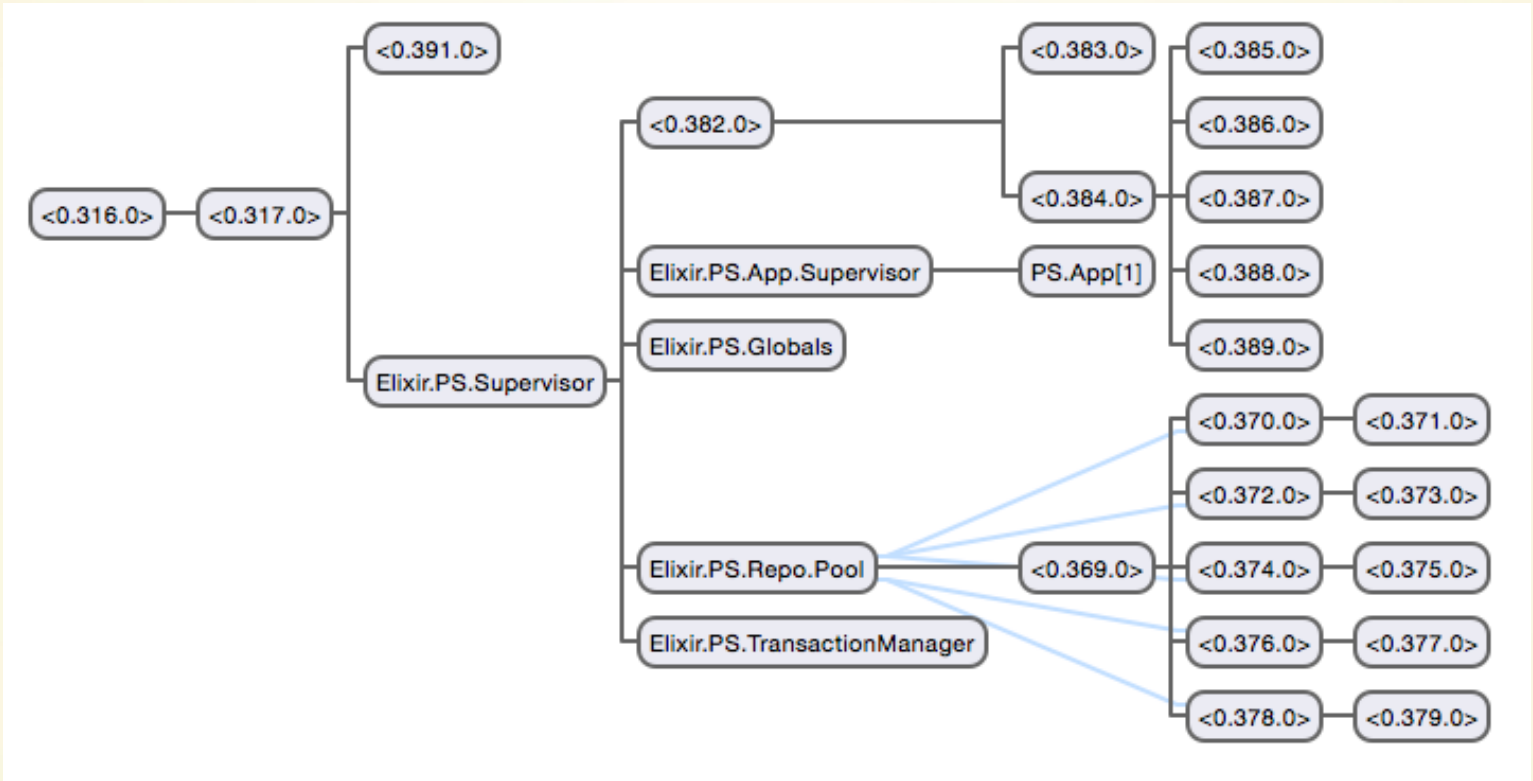
Elixir is concurrent

- All code runs inside a process
- Processes are lightweight and isolated, they don't share any state
- Processes communicate by message passing

Spawning processes 1/2

```
iex(1)> parent = self()
#PID<0.53.0>

iex(2)> Enum.each(1..3, fn n ->
...(2)>   spawn_link(fn -> send(parent, {:hello, self(), n}) end)
...(2)> end)
:ok

iex(3)> flush
{:hello, #PID<0.68.0>, 1}
{:hello, #PID<0.69.0>, 2}
{:hello, #PID<0.70.0>, 3}
```

Spawning processes 2/2



Elixir is functional

- Pattern matching
- Immutable data
- Composition over encapsulation
- Declarative control flow

Pattern matching 1/5

Asserts that two expressions are equal

```
a = 1
1 = a

2 = a  #=> ** (MatchError) no match of right hand side value: 1


"hello" <> rest = "hello world"
rest  #=> " world"
```

Pattern matching 2/5

Supports arbitrarily complex patterns

```
{:type, "error", %{
  code: code, reason: _, backtrace: [latest_call | _]
}} =
{:type, "error", %{
  reason: "bad argument", code: 13, backtrace: [
    {String, :length, {"string.ex", 13}}, ...
  ]}
}

code          #=> 13
latest_call   #=> {String, :length, {"string.ex", 13}}
```

Pattern matching 3/5

Facilitates ***intentional programming***

```
{:ok, file} = File.open "rtfm.txt"

#=> ** (MatchError) no match of right hand side value:
#                   {:error, :enoent}
```

Pattern matching 4/5

Doesn't take control away from
the programmer

```
case File.open("rtfm.txt") do
  {:ok, file} -> ...
  {:error, :enoent} -> ...
end
```

Pattern matching 5/5

Extensively used in function definitions
together with ***guard expressions***

```
defp hex_to_dec(n) when n in ?A..?F, do: n - ?A + 10
defp hex_to_dec(n) when n in ?a..?f, do: n - ?a + 10
defp hex_to_dec(n) when n in ?0..?9, do: n - ?0

defp hex_to_dec(_n), do: throw(:malformed_uri)
```

Immutable values 1/5

Arbitrary hierarchical structures are values, just like numbers

```
[1, 2, 3]
{:atom, "value"}

m = %{foo: "bar"}
m1 = Map.put(m, :baz, "quux")

m   #=> %{foo: "bar"}
m1  #=> %{baz: "quux", foo: "bar"}
```

Immutable values 2/5

Encourages programming "data processing pipelines"

```
nodes
|> Enum.map(&String.strip/1)
|> Enum.reject(&( &1 == "" ))
|> Enum.map(fn node_name ->
  [name, _] = String.split(node_name, "@")
  {name, String.to_atom(node_name)}
end)

nodes = ["foo@localhost", "", "bar@example.com"]
parse_node_list(nodes)
#=> [ {"foo", :"foo@localhost"}, {"bar", :"bar@example.com"} ]
```

Immutable values 3/5

Enforces preference of composition
over incapsulation

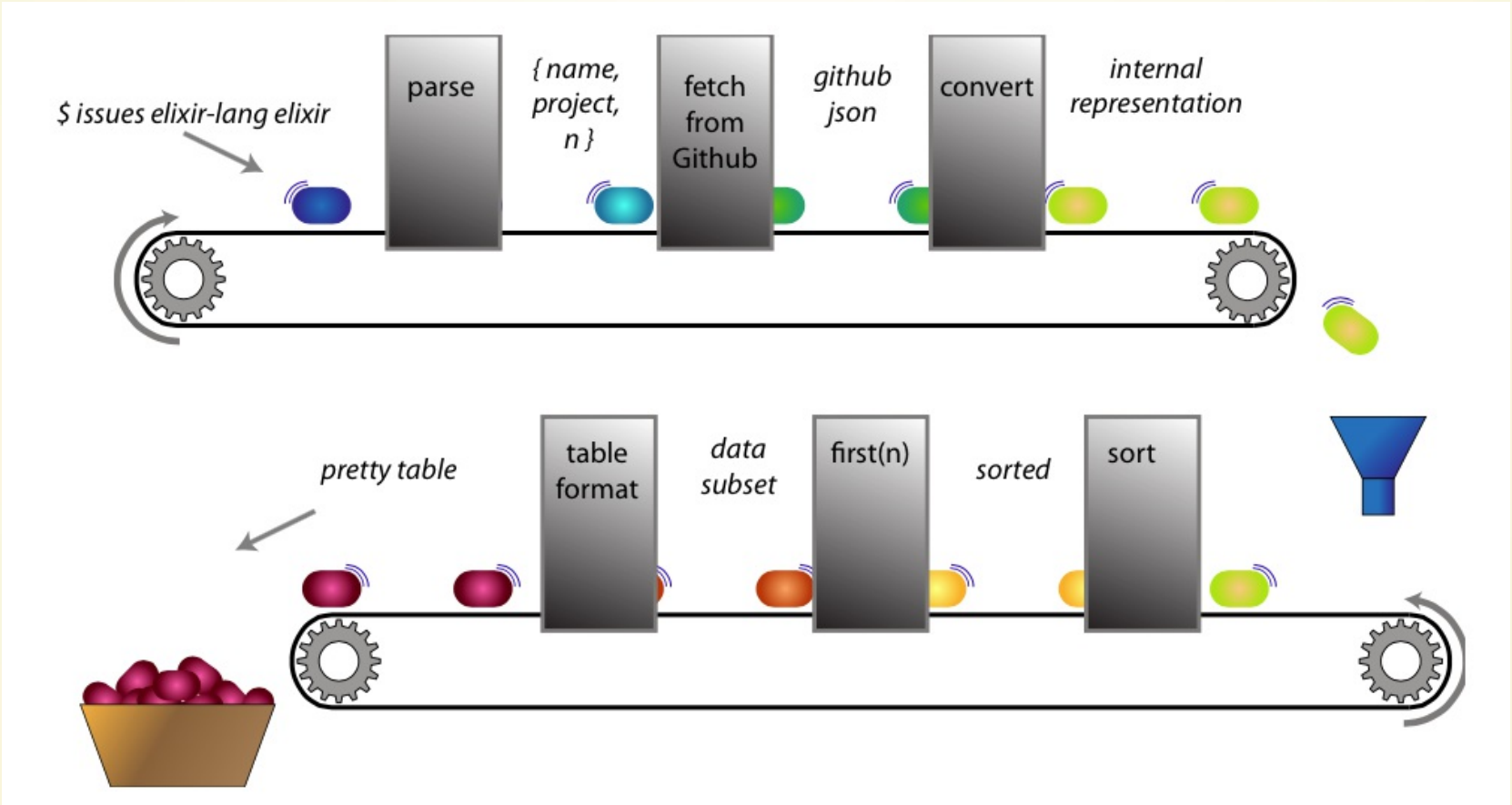
```
[foo: 1, bar: 2, baz: "three"] |> Enum.into(HashDict.new)

# %{__struct__: HashDict,
#   root: {[], [], {:bar, 2, {[], [:baz | "three"],
#    [], [], [], [], [], []}}, [],
#    [], [], [], [:foo | 1]}, size: 3}
```

Immutable values 4/5

Another example of a "pipeline"

```
def process({user, project, count}) do
  Issues.GithubIssues.fetch(user, project)
  |> decode_response
  |> convert_to_list_of_hashdicts
  |> sort_into_ascending_order
  |> Enum.take(count)
  |> print_table_for_columns(["number", "created_at", "title"])
end
```

Immutable values 5/5



Source: Programming Elixir

Controlled state 1/3

It is often obvious from the code, when it changes

```
def handle_call({:send_msg, msg}, _from,
        %State{socket: sock, chan: chan, sent_count: msg_count} = state)
do
  :ok = :gen_tcp.send(sock, ["PRIVMSG", " ", chan, " :", msg, "\r\n"])
  new_state = %Struct{state | sent_count: msg_count+1}
  {:reply, :ok, new_state}
end
```

Controlled state 2/3

Allows for straightforward code swapping

```
def code_change(_oldVsn, state, _extra) do
  new_state = convert_to_new_representation(state)
  {:ok, new_state}
end
```

Controlled state 3/3

Processes are just like objects

```
{:ok, conn} = DB.new_connection(url)
results = DB.query(conn, "...")

# Change the process' internal state
DB.set_timeout(conn, 1000)
DB.get_timeout(conn)       #=> 1000
```

And there's more

- Polymorphism via protocols
- Lean standard library (with OTP additions)
- Meta-programming via macros
- mix: build tool and project management tool
- hex.pm: standard ecosystem of libraries
- 1st-class Unicode support; lazy streams; extensible; interactive development; Erlang compatibility; etc.

Protocols

```
defprotocol JSON.Decoder do
  def from_json(data)
end

defprotocol JSON.Encoder do
  def to_json(self, options)
end

JSON.encode!(dict / list / record)
JSON.decode!(json)
```

```
defimpl JSON.Encoder, for: Anything do
    def to_json(...), do: ...
end

defimpl JSON.Decoder, for: Anything do
    def from_json(...), do: ...
end

JSON.encode!(anything)
JSON.decode!(json, as: anything)
```

Meta-programming 1/2

```
defmodule Pipespect do
  defmacro __using__(_) do
    quote do
      import Kernel, except: [{:|>, 2}]
      import unquote(__MODULE__), only: [{:|>, 2}]
    end
  end

  import Kernel, except: [{:|>, 2}]
  defmacro first |> rest do
    inspect = quote do: IO.inspect
    stages =
      Enum.intersperse(Enum.map(Macro.unpipe(rest), fn {x, _} -> x end), inspect)
    quote do
      unquote(first) |> unquote(rebuild_pipe(stages ++ [inspect]))
    end
  end

  defp rebuild_pipe([h]) do
    h
  end

  defp rebuild_pipe([h|t]) do
    {:|>, [], [h, rebuild_pipe(t)]}
  end
end
```

Meta-programming 2/2

```
import Enum

[1,2,3] |> reverse |> map(& &1*&1) |> join(" . ")
# output: none

use Pipespect
[1,2,3] |> reverse |> map(& &1*&1) |> join(" . ")
# output:
#   [3, 2, 1]
#   [9, 4, 1]
#   "9 . 4 . 1"
```

Standard library

- Process
- Protocol
- Range
- Record
- Regex
- Set
- Stream
- String
  - at/2
  - capitalize/1
  - chunk/2
  - codepoints/1
  - contains?/2
  - downcase/1
  - duplicate/2
  - ends_with?/2
  - first/1
  - graphemes/1
  - last/1
  - length/1
  - ljust/2
  - ljust/3
  - lstrip/1
  - lstrip/2
  - match?/2

String

Summary | Functions | Types

A String in Elixir is a UTF-8 encoded binary.

Codepoints and graphemes

The functions in this module act according to the Unicode Standard, version 6.3.0.As per the standard, a codepoint is an Unicode Character, which may be represented by one or more bytes. For example, the character "é" is represented with two bytes:

```
iex> byte_size("é")
2
```

However, this module returns the proper length:

```
iex> String.length("é")
1
```

Furthermore, this module also presents the concept of graphemes, which are multiple characters that may be "perceived as a single character" by readers. For example, the same "é" character written above could be represented by the letter "e" followed by the accent´:

```
iex> string = "\x{0065}\x{0301}"
iex> byte_size(string)
3
iex> String.length(string)
1
```

Although the example above is made of two characters, it is perceived by users as one.

Graphemes can also be two characters that are interpreted as one by some languages. For example, some languages may consider "ch" as a grapheme. However, since this information depends on the locale, it is not taken into account by this module.

In general, the functions in this module rely on the Unicode Standard, but does not contain any of the locale specific behaviour.

More information about graphemes can be found in the Unicode Standard Annex #29. This current Elixir version implements Extended Grapheme Cluster algorithm.

Mix tool

```
$ mix help
mix archive.build      # Archive this project into a .ez file
mix archive.install    # Install an archive locally
mix bench              # Benchmark your code
mix bench.cmp          # Compare benchmark snapshots
mix bench.graph        # Produce an HTML page with graphs built from given snapshots
mix compile            # Compile source files
mix deps               # List dependencies and their status
mix deps.get           # Get all out of date dependencies
mix deps.update        # Update the given dependencies
mix dialyzer           # Runs dialyzer with default or project-defined flags.
mix docs               # Generate HTML documentation for the project
mix escript.build      # Builds an escript for the project
mix new                # Create a new Elixir project
mix test               # Run a project's tests
iex -S mix             # Start IEx and run the default task
```

Elixir is also ...

… fault-tolerant

- No ***defensive conding***
- Unexpected errors cause processes to crash
- A process can watch another process, detect the crash, and restart it
- Supervisors and workers

... distributed

- A process is identified by a PID
- Messages are sent to PIDs, regardless of the process' physical location
- Easy to set up communication between machines or even execute code remotely

... scalable

- BEAM is the Erlang virtual machine
- Optimized for **minimum latency**
- It has been used to build systems with massive loads
- It has been used in embedded systems
- When Node.js fails under load, BEAM keeps running

... maintainable

- OTP provides a framework for building reusable components
- BEAM has lots of inbuilt facilities for live introspection of a running system
- Live code reloading

Why not Erlang?

- New features not found in Erlang (protocols, structs, sigils, macros, OTP additions, nicer string handling)
- Out-of-the-box tooling
- The language is extensible
- The community is more focused on experimentation and innovation
- It just feels good

Where to go next

elixir-lang.org

Elixir wiki

Also take a look at how cool BEAM is

www.youtube.com/watch?v=xrIjfIjssLE

Thank you!

Questions?

@true_droid

github.com/alco