CSC 533 Presentation: Elixir

Jerome Bustarga, Cameron Kelly, Logan Olson

Elixir Presentation Overview

01

History of Elixir

02

Design Goals of Elixir

03

Elixir Sample Code

04

Real World Applications

05

Elixir References

elixir

History of Elixir

José Valim

01 Origins

- Created by José Valim, a Ruby on Rails core contributor
- Built on top of the Erlang VM (BEAM)
- Inherited fault tolerance of Erlang

02 Early Development

- Introduced package manager hex (like pip for Python or npm for JS) and built tool Mix
- Appealed to devs who didn't want to sacrifice productivity for performance

03 Present Elixir

- Stable 1.0 release in 2014
- Rise of the Phoenix framework (for web-dev)
- Steady releases have cultivated the Elixir ecosystem and community

Erlang Virtual Machine (BEAM)

- Developed by Ericsson in the 80s to handle payphone calls
- Designed for massive concurrency
- Built with fault isolation
- Engineered for 99.9999% uptime
- Elixir inherits all of the benefits from the BEAM
- So it gets decades of telecom-grade reliability for free

What Were Elixir's Goals?

- Functional programming language built on BEAM
- Data transformation over state management
- Concurrency and Scalability
- Fault Tolerance
- Extensibility
- Fully Erlang Compatible

Similarities to Clojure

- Both functional languages
- Both run on VMs
- Data is immutable by default
- Both have first class functions
- Clojure can call Java, Elixir can call Erlang

Differences from Clojure

```clojure
(defn add [a b]
  (+ a b))
```

```elixir
defmodule Math do
  def add(a, b) do
    a + b
  end
end
```

- Syntax
  - Elixir: Ruby, Clojure: Lisp
- Concurrency
  - Elixir: actor model, Clojure: STM (shared memory)
- Fault Tolerance
  - Elixir: built-in supervisors
- Pattern matching
- Target use case
  - Elixir: distributed systems, Clojure: data processing
- Error philosophy

Features

- Data Types
- Variables, Bindings, & Immutability
- Control Structures
- Functions & Abstraction
- Pattern Matching
- Pipe Operator
- Processes and Message Passing
- "Let it Crash" + OTP Supervisors
- Memory Management

Data Types

Primitive types

- Integers, floats, booleans, atoms strings

Composite types

- Tuple, map, keyword list, struct

```
list = [1, 2, 3]
hd(list)          #=> 1 (the element, not a list)
tl(list)          #=> [2, 3]
[0 | list]        #=> [0, 1, 2, 3]
Enum.at(list, 0)  #=> 1
```

```
opts = [timeout: 5000, retries: 3]
opts[:timeout]    #=> 5000
```

Variables, Bindings, & Immutability

- Dynamic typing (checked at runtime)
- Data is immutable
- Variables can be rebound to new values
- Static scoping (no global mutable state)

```
iex(1)> x = 10
10
iex(2)> x = "hello"
"hello"
iex(3)> x
"hello"
```

```
iex(1)> x = 10
10
iex(2)> if true do
          x = 20
          IO.puts(x)
        end
20
:ok
iex(3)> x
10
```

Control Structures

- Case: pattern match on a value
- Cond: if-else chain
- If/Unless: simple conditionals
- No for/while loops (can't mutate a counter like i++)
  - Use recursion, Enum module, or comprehensions

```
case Enum.at(list, 2) do
  1  -> "one"
  3  -> "three"
  _  -> "catch all"
end

cond do
  x > 10 -> "big"
  x > 5  -> "medium"
  true   -> "small"
end
```

Functions & Abstraction

- Named functions (def) inside modules, anonymous (fn)
- No explicit return, last expression is the value
- Multiple function heads via pattern matching
- No classes: modules, protocols, behaviours instead

```
defmodule Math do
  def factorial(0), do: 1
  def factorial(n), do: n * factorial(n - 1)
end

iex> Math.factorial(5)
120
```

Pattern Matching

Concept

- = is the matching operator, NOT the assignment operator
- Variables bind to values through matching

```
iex(1)> {a, b} = {1, 2}
{1, 2}
iex(2)> a
1
iex(3)> b
2
```

Why it matters

- Extracts values easily
- Reduces need for complex conditionals
- Makes code more readable

```
iex(1)> {:ok, result} = {:ok, 42}
{:ok, 42}
iex(2)> {:ok, result} = {:error, 42}
** (MatchError) no match of right hand side value:

    {:error, 42}

    (stdlib 7.3) erl_eval.erl:672: :erl_eval.expr/6
    iex:2: (file)
```

Pipe Operator |>

- Pass the result of one function as the first argument to the second
- Reads left to right

```
iex(1)> "elixir" |> String.ends_with?("ixir")
true
iex(2)> "Elixir rocks" |> String.upcase() |> String.split()
["ELIXIR", "ROCKS"]
iex(3)> "Elixir rocks" |> String.split()
["Elixir", "rocks"]
```

Processes and Message Passing

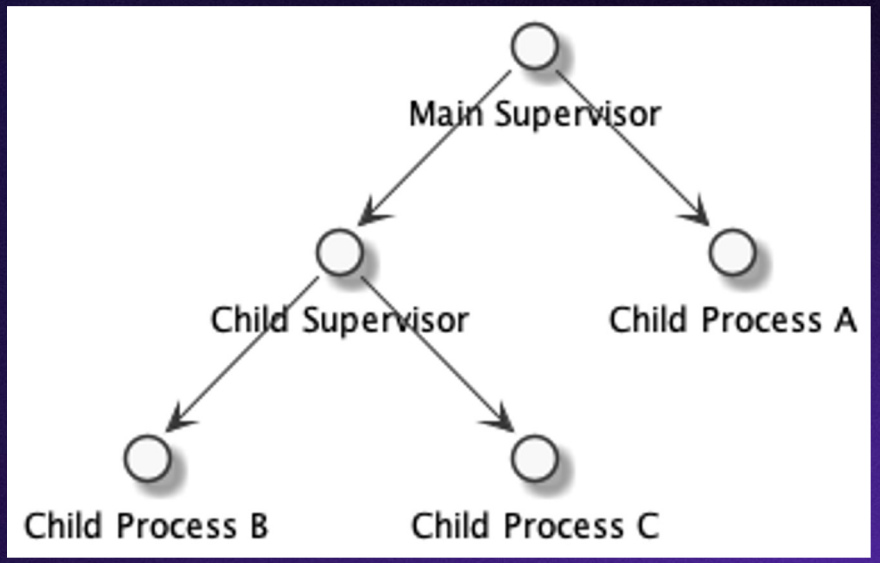
- Processes are not OS threads
- Millions of processes simultaneously
- Each process has its own memory
- Crash in one process does not affect others

```
iex(1)> pid = spawn(fn ->
          receive do
            {:hello, msg} -> IO.puts("Got: #{msg}")
          end
        end)
#PID<0.105.0>
iex(2)> send(pid, {:hello, "world"})
Got: world
{:hello, "world"}
iex(3)>
```

“Let it Crash” + OTP Supervisors

“Let it crash” Philosophy

- Serves to handle crashes rather than prevent them
- Processes are expected to fail
  - These failures should be isolated
  - Don't need defensive programming everywhere



OTP Supervisors

- Supervisors monitor processes
  - Defines restart strategies
  - Restart only what fails, not entire system
    - Only failed processes cease execution

Memory Management

```
iex(1)> list1 = [1, 2, 3, 4]
[1, 2, 3, 4]
iex(2)> list2 = [0 | list1]
[0, 1, 2, 3, 4]
```

01

Garbage Collection

- Each process has its own heap and garbage collector
- Prevents system-wide garbage collection pauses

02

Process Isolation

- Processes don't share resources
- Communication via message passing

03

Efficient Memory Usage

- Immutability allows for structure sharing
- E.g. lists reuse existing structure instead of copying

04

Process Life Cycle

- Memory is reclaimed when process terminate; lightweight processes are cheap to create and destroy

Who is using it?

Sources

- https://medium.com/wttj-tech/the-3-features-that-will-make-you-fall-for-elixir-598c9a31172d
- https://hexdocs.pm/elixir/introduction.html
- https://elixir-lang.org/
- https://en.wikipedia.org/wiki/Elixir_(programming_language)
- https://youtu.be/lxYFOM3UJzo?si=zNFij0RgUlOZEGUc
- https://hexdocs.pm/elixir/basic-types.html
- https://elixir-lang.org/learning.html
- https://youtu.be/Q0Z1jqv6LW0?si=DjKapDUI5FVe3ik9
- https://hexdocs.pm/phoenix/up_and_running.html
- https://elixir-companies.com/en/companies
- https://hexdocs.pm/phoenix/overview.html
- https://medium.com/wttj-tech/the-3-features-that-will-make-you-fall-for-elixir-598c9a31172d
- https://elixirforum.com/t/what-are-the-most-important-features-elixir-has-for-you-and-why/26850/4
- https://akoutmos.com/post/betting-on-elixir/
- https://enterpriseviewpoint.com/ericsson-plans-to-lay-off-8-of-its-global-employees/
- https://serokell.io/blog/introduction-to-elixir
- https://elixirschool.com/en/lessons/basics/pipe_operator
- https://commons.wikimedia.org/wiki/File:Spotify_logo_without_text.svg
- https://dribbble.com/shots/5240476-Bleacher-Report-Logo

Sources Cont.

- https://icon-icons.com/icon/pepsi-logo/168910
- https://global.toyota/en/mobility/toyota-brand/features/emblem/
- https://commons.wikimedia.org/wiki/File:Pinterest-logo.png
- https://www.vecteezy.com/vector-art/6892625-discord-logo-icon-editorial-vector

Thank you!