Programming with Elixir

Pramode C.E

http://pramode.net

February 16, 2015

Workshop Plan

Here is what we will do:

- Get some general ideas about the Erlang platform.
- Learn the basics of Elixir.
- Write some simple programs in Elixir.

The Erlang platform

- First version developed by Joe Armstrong and others at Ericsson in 1986.
- Released as Open Source in 1998.
- Designed for developing telephony applications where reliability is critical.
- Ericsson's AXD301 has over 10 lakh lines of Erlang code and is said to have "nine-nines" availability (31.5ms downtime/year)!

The Erlang platform

"The network performance has been so reliable that there is almost a risk that our field engineers do not learn maintenance skills." – Bernt Nilsson

The Erlang Platform

The Erlang Platform

Key features of Erlang:

- Concurrency
- Fault tolerance
- Distributed Computing
- Functional Programming

Erlang at Whatsapp

- Facebook deal worth 19 billion dollars.
- 55 employees at the time of the deal.
- 600 million (60 crore) active users.
- A tweet on Dec 31, 2013 said: 18 billion messages/day.
- Backend uses: FreeBSD and Erlang.

More info: http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html
Case studies:
https://www.erlang-solutions.com/resources/case-studies

The Elixir Story

- Developed by Jose Valim (2012).
- Main goals: more productivity + modern syntax + ALL the benefits of Erlang.
- Compiles to BEAM (the Erlang VM) byte code - so can use ALL the features of Erlang without any runtime impact.
- Major features: metaprogramming with macros, polymorphism via protocols.
- Great tooling out-of-the-box: build system, package manager, test framework.

The Elixir Story

Why learn Elixir?

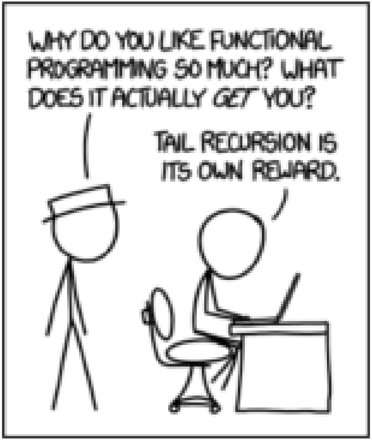
Why Learn Elixir/Erlang?

Why learn Elixir?

Or, for that matter, Haskell, Ocaml, Clojure, Scala, Go ...

- Learning new ways to think about problem solving
- Concurrent/Distributed programming is the future
- Functional programming is gaining more acceptance
- The Python paradox (2004): paulgraham.com/pypar.html. "The language to learn, if you want to get a good job, is a language that people don't learn merely to get a good job".

Why learn Elixir?


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

Getting Started with Elixir

Use the REPL (type "iex" at the command prompt) to do some simple experiments:

```
iex(1)> a = 1
1
iex(2)> b = 2
2
iex(3)> a + b
3
iex(4)> 5 / 2
2.5
iex(5)> div(5, 2)
2
iex(6)> rem(5,2)
1
iex(7)>
```

The "match" operator

The operator "=" is a bit weird!

```
iex(1)> a = 1
1
iex(2)> 1 = a
1
iex(3)> 2 = a
** (MatchError) no match of right hand side value: 1

iex(4)> a = 2
2
iex(5)>
```

Basic Types

- 1, 0x1f, 0o12, 0b101 : Integer
- 1.0 : Float
- true, false : Boolean
- :foo, :bar : atom/symbol
- "hello" : string
- [1,2,3]: list
- {1,2,3} : tuple

Basic Types and Operations

```
iex(1)> true == false
false
iex(2)> is_boolean(false)
true
iex(3)> is_atom(:foo)
true
iex(4)> x = 2
2
iex(5)> "foo #{x}"
"foo 2"
iex(6)> "foo" <> "bar"
"foobar"
iex(7)> String.length("foo")
3
iex(8)>
```

Basic Types and Operations

```
iex(1)> String.upcase("foo")
"FOO"
iex(2)> true and false
false
iex(3)> 1 or true
** (ArgumentError) argument error: 1

iex(4)> not true
false
iex(5)>
```

List/Tuple basics

```
iex(1)> a = [1,2,3]
[1,2,3]
iex(2)> a ++ [4,5,6]
[1,2,3,4,5,6]
iex(3)> a
[1,2,3]
iex(4)> a -- [2]
[1,3]
iex(5)> length(a)
2
iex(6)> hd(a)
1
iex(7)> tl(a)
[2,3]
iex(8)>
```

List/Tuple basics

```
iex(8)> [10 | a]
[10,1,2,3]
iex(9)> [10,11,12|a]
[10,11,12,1,2,3]
iex(10)> b = {10, 20, 30}
{10,20,30}
iex(11)> tuple_size(b)
3
iex(12)> elem(b, 0)
10
iex(13)> put_elem(b,1,100)
{10,100,30}
iex(14)> b
{10,20,30}
iex(15)>
```

Maps

```
iex(1)> d = %{"foo" => 10, "bar" => 20, "abc" => 30}
%{"abc" => 30, "bar" => 20, "foo" => 10}
iex(2)> d["foo"]
10
iex(3)> Dict.values(d)
[30, 20, 10]
iex(4)> Dict.has_key?(d, "bar")
20
iex(5)>
```

Maps

```
iex(1)> m = %{"a" => 1, "b" => 2}
%{"a" => 1, "b" => 2}
iex(2)> %{m | "a" => 10, "b" => 20}
%{"a" => 10, "b" => 20}
iex(3)> m
%{"a" => 1, "b" => 2}
iex(4)> Dict.put_new(m, "c", 3)
%{"a" => 1, "b" => 2, "c" => 3}
iex(5)>
```

Sets

```
iex(1)> HashSet.new
#HashSet<[]>
iex(2)> s1 = Enum.into(1..10, HashSet.new)
#HashSet<[7, 2, 6, 3, 4, 1, 5, 9, 10, 8]>
iex(3)> Set.member?(s1, 3)
true
iex(4)> s2 = Enum.into(5..13, HashSet.new)
#HashSet<[7, 13, 6, 5, 9, 11, 10, 12, 8]>
iex(5)> Set.union(s1,s2)
#HashSet<[7, 13, 2, 6, 3, 4, 1, 5, 9, 11, 10, 12, 8]>
iex(6)>
```

Pattern matching

```
iex(1)> a = [1,2,3]
[1,2,3]
iex(2)> [p, q, r] = a
[1,2,3]
iex(3)> p
1
iex(4)> q
2
iex(5)> r
3
iex(6)> [1,2,3] = a
[1,2,3]
iex(7)> [1,2] = a
** (MatchError) no match of right hand side value: [1, 2, 3]
iex(8)>
```

Pattern matching

```
iex(8)> [p, q] = [1, [2,3]]
[1,[2,3]]
iex(9)> q
[2,3]
iex(10)> [1, r] = [1, [2,3]]
[1, [2,3]]
iex(11)> r
[2,3]
iex(12)> [f | g] = [1,2,3,4]
[1,2,3,4]
iex(13)> f
1
iex(14)> g
[2,3,4]
iex(15)>
```

Pattern matching

```
iex(16)> [m | _] = [1,2,3,4]
[1,2,3,4]
iex(17)> m
1
iex(18)> [t, t] = [1, 2]
* (MatchError) no match of right hand side value: [1, 2]

iex(19)> {i, j} = {1, 2}
{1,2}
iex(20)> i
1
iex(21)> j
2
iex(22)>
```

Anonymous Functions

```
iex(1)> sqr = fn x -> x*x end
#Function<6.90072148/1 in :erl_eval.expr/5>
iex(2)> sqr.(10)
100
iex(3)> add = fn a b -> a + b end
#Function<12.90072148/2 in :erl_eval.expr/5>
iex(4)> add.(10,20)
30
iex(5)> greet = fn -> IO.puts "hello"
#Function<12.90072148/2 in :erl_eval.expr/5>
iex(6)> greet.()
hello
:ok
iex(7)>
```

Anonymous Functions

```
iex(1)> x = 1
1
iex(2)> f = fn a -> a + x end
#Function<6.90072148/1 in :erl_eval.expr/5>
iex(3)> f.(10)
11
iex(4)> x = 2
2
iex(5)> x
2
iex(6)> f.(10)
11
iex(7)>
```

Anonymous Functions

```
iex(1)> sqr = &(&1 * &1)
#Function<6.90072148/1 in :erl_eval.expr/5>
iex(2)> sqr.(10)
100
iex(3)> sum = &(&1 + &2)
&:erlang.+/2
iex(4)> sum.(1,2)
3
iex(5)> f = &IO.puts/1
&IO.puts/1
iex(6)> f.("hello")
hello
:ok
iex(7)>
```

Anonymous Functions

```
iex(1)> sqr = &(&1 * &1)
#Function<6.90072148/1 in :erl_eval.expr/5>
iex(2)> cube = &(&1 * &1 * &1)
#Function<6.90072148/1 in :erl_eval.expr/5>
iex(3)> f = fn (f, g, x) -> f.(g.(x)) end
#Function<18.90072148/3 in :erl_eval.expr/5>
iex(4)> f.(sqr, cube, 2)
64
iex(5)> Enum.map([1,2,3,4], &(&1*&1))
[1,4,9,16]
iex(6)> f = fn x -> fn y -> x + y end end
#Function<6.90072148/1 in :erl_eval.expr/5>
iex(7)> g = f.(10)
#Function<6.90072148/1 in :erl_eval.expr/5>
iex(8)> g.(20)
30
iex(9)>
```

Anonymous Functions

```
iex(1)> f = fn
...(1)>  0 -> "hello"
...(1)>  x -> x + 1
...(1)> end
#Function<6.90072148/1 in :erl_eval.expr/5>
iex(2)> f.(0)
"hello"
iex(3)> f.(10)
11
iex(4)> f.(20)
21
iex(5)>
```

Note: the number of parameters in each clause of the function definition should be same.

Modules and Functions

```
iex(1)> defmodule Foo do
...(1)>    def sqr(x) do
...(1)>        x*x
...(1)>    end
...(1)> end
iex(2)> Foo.sqr(10)
100
iex(3)>
```

Modules and Functions

A slightly different syntax:

```
iex(1)> defmodule Foo do
...(1)>    def sqr(x), do: x*x
...(1)> end
iex(2)> Foo.sqr(10)
100
iex(3)>
```

This syntax is preferred for one-line functions.

Modules and Functions

```
iex(1)> defmodule Math do
...(1)>   def factorial(0), do: 1
...(1)>   def factorial(n), do: n * factorial(n-1)
...(1)> end
iex(2)> Math.factorial(0)
1
iex(3)> Math.factorial(4)
24
iex(4)>
```

Note: The order of the clauses is important.

Modules and Functions

Use of "guard" clauses:

```
iex(1)> defmodule Math do
...(1)>    def factorial(0), do: 1
...(1)>    def factorial(n) when n > 0 do
...(1)>        n*factorial(n-1)
...(1)>    end
...(1)> end
iex(2)> Math.factorial(4)
24
iex(3)>
```

Modules and Functions

```
iex(1)> defmodule Foo do
...(1)>    def fun(p1 \\10, p2 \\20), do: [p1,p2]
...(1)> end
iex(2)> Foo.fun()
[10,20]
iex(3)> Foo.fun(1)
[1,20]
iex(4)> Foo.fun(1,2)
[1,2]
iex(5)>
```

Modules and Functions - The Amazing Pipe Operator

```
iex(1)> a = Enum.map(1..10, &(&1*&1))
iex(2)> a
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
iex(3)> b = Enum.filter(a, &(&1<40))
iex(4)> b
[1, 4, 9. 16, 25, 36]
iex(5)> 1..10 |> Enum.map(&(&1*&1)) |> Enum.filter(&(&1<40)
[1, 4, 9, 16, 25, 36]
iex(6)>
```

val |> f(a,b) is same as f(val, a, b)

Recursive List Processing

```
iex(1)> defmodule MyList do
...(1)>    def len([]), do: 0
...(1)>    def len([head | tail]), do: 1 + len(tail)
...(1)> end
iex(2)> MyList.len([10,20,30])
3
iex(3)>
```

Recursive List Processing

A small exercise:

Sum of elements

Write a function which will sum all the elements of a list of integers.

Recursive List Processing

Another exercise:

Destutter a list

Write a function to eliminate adjacent repeated elements from a list.

Thinking Functionally

Problem: Find the N most commonly occuring words in a text file. We will do this by writing a few simple functions and composing them with Elixir "pipes".

The List module

```
iex(1)> List.flatten([[1,2],[[3]]])
[1, 2, 3]
iex(2)> List.zip([[1,2,3], [4,5,6]])
[{1, 4), {2, 5}, {3, 6}]
iex(3)> List.keyfind([{10,"foo}, {1,"abc"}, {2,"def’}],
...(3)>              "abc", 1)
{1, abc}
iex(4)>
```

Control Structures

```
iex(1)> 1 + if true, do: 10, else: 20
11
iex(2)> if true do
...(2)>   10
...(2)> else
...(2)>   20
...(2)> end
10
iex(3)>
```

Control Structures

```
iex(1)> i = 1
1
iex(2)> cond do
...(2)>      i == 10 -> "a"
...(2)>      i == 20 -> "b"
...(2)>      i == 1  -> "c"
...(2)> end
"c"
iex(3)>
```

Control Structures

```
iex(1)> case File.open("elixir.tex") do
...(1)>     {:ok, file} -> IO.puts IO.read(file, :line)
...(1)>     {:error, reason} -> "failed; reason: #{reason}"
...(1)> end
\documentclass {beamer}
:ok
iex(2)>
```

Concurrent Programming

"Traditional" concurrent programming:

- Threads
- Shared data
- Mutexes, Semaphores
- Deadlocks, race conditions

Concurrent Programming

Erlang concurrency model

Light-weight VM managed "actors" which do not share any state and communicate with each other through message passing.

Concurrent Programming

```
iex(1)> pid = spawn(fn -> IO.puts "hello" end)
hello
#PID<0.56.0>
iex(2)> Process.alive? pid
false
iex(3)>
```

Concurrent Programming

Let's try it in a slightly different way; first, create a file "a.ex":

```
defmodule Foo do
    def loop(msg) do
        IO.puts msg
        :timer.sleep(1000)
        loop(msg)
    end
end
```

Concurrent Programming

Now we will use "iex" to test the code:

```
iex(1)> c("a.ex")
[Foo]
iex(2)> pid1 = spawn(Foo, :loop, ["hello..."])
hello...
#PID<0.72.0>
hello...
hello...
iex(3)> Process.exit(pid1, :kill)
true
iex(4)>
```

Concurrent Programming

A task communicating with itself:

```
iex(1)> send self, 10
10
iex(2)> receive do
...(2)>     x -> x + 1
...(2)> end
11
iex(3)> send self, {:ok, "hello"}
{:ok,"hello"}
iex(4)> p = receive do
...(4)>      {:ok, msg} -> msg
...(4)> end
"hello"
iex(5)> p
"hello"
iex(6)>
```

Concurrent Programming

A task communicating with another:

```
iex(1)> self
#PID<0.54.0>
iex(2)> spawn(fn -> IO.puts (inspect self) end)
#PID<0.57.0>
#PID<0.57.0>
iex(3)> me = self
iex(4)> spawn(fn -> send me, 10 end)
#PID<0.61.0>
iex(5)> p = receive do x -> x end
10
iex(6)> p
10
iex(7)>
```

Concurrent Programming

Bi-directional message passing:
Let’s create a file "b.ex" containing:

```
defmodule Foo1 do
  def fun do
    receive do
      {sender, msg} ->
          send(sender, {:ok, "Got the message: #{msg}"})
    end
  end
end
```

Concurrent Programming

Now, lets try some experiments with "iex":

```
iex(1)> c("b.ex")
[Foo1]
iex(2)> pid = spawn(Foo1, :fun, [])
#PID<0.62.0>
iex(3)> send pid, {self, "Hello!"}
{#PID<0.54.0>, "Hello!"}
iex(4)> receive do
...(4)>     {:ok, reply} -> reply
...(4)> end
"Got the message: Hello!"
iex(5)>
```

Concurrent Programming

Using Timeouts:

```
iex(1)> receive do
...(1)>     x -> x + 1
...(1)>     after 500 -> "timeout"
...(1)> end
"timeout"
iex(2)>
```

Concurrent Programming

Erlang processes are light weight: 309 words (1236 bytes) per process. Let's verify this by spawning 100000 new tasks and checking system memory usage with "htop":

```
iex(1)> Enum.each(1..100000,
...(1)>          fn _ -> spawn(
...(1)>          fn -> :timer.sleep(1000000) end) end)
:ok
iex(2)>
```

Concurrent Programming

Process monitoring.
Create a file called "monitor.ex":

```
defmodule Monitor1 do
    def foo do
      :timer.sleep(5000)
    end

    def run_foo do
        pid = spawn_monitor(Monitor1, :foo, [])
        IO.puts inspect pid
        receive do
            msg -> "msg = #{inspect msg}"
        end
    end
end
```

Concurrent Programming

Let's now test this out with "iex":

```
iex(1)> c("monitor.ex")
[Monitor1]
iex(2)> Monitor1.run_foo
"msg = {:DOWN, #Reference<0.0.0.135>,
:process, #PID<0.62.0>, :normal}"
iex(3)>
```

Concurrent Programming

A "chain" of processes:

```
defmodule Chain2 do
    def foo next_pid do
        receive do
            n -> send next_pid, n+1
        end
    end
    def run do
        last = Enum.reduce(1..100000, self,
                    fn (_, send_to) ->
                        spawn(Chain2, :foo, [send_to])
                    end)
        send last, 0
    end
end
```

Concurrent Programming

Try out the code in "iex":

```
iex(1)> c("chain2.ex")
[Chain2]
iex(2)> Chain2.run
0
iex(3)> receive do x -> x end
100000
iex(4)>
```

Concurrent Programming

A "parallel map" implementation. Contents of file "pmap.ex" shown below:

```
defmodule Parallel do
    def pmap(collection, fun) do
        me = self
        collection
        |> Enum.map(fn (elem) ->
                spawn(fn -> send(me, {self, fun.(elem)}) end
             end)
        |> Enum.map(fn (pid) ->
                receive do {^pid, result} -> result end
             end)
    end
end
```

Concurrent Programming

We will test out the code with "iex":

```
iex(1)> c("pmap.ex")
[Parallel]
iex(2)> Parallel.pmap(1..5, fn x -> x * x end)
[1, 4, 9, 16, 25]
iex(3)>
```

Distributed Computing

Start two instances of the Erlang VM on two terminals by typing "iex –name one@127.0.0.1" and "iex –name two@127.0.0.1"

```
iex(one@127.0.0.1)1> Node.self
:"one@127.0.0.1"
iex(one@127.0.0.1)2> Node.list
[]
iex(one@127.0.0.1)3> Node.ping(:"two@127.0.0.1")
:pong
iex(one@127.0.0.1)4> Node.list
[:"two@127.0.0.1"]
iex(one@127.0.0.1)5> f = fn ->
...(one@127.0.0.1)5>    IO.puts inspect Node.self end
#Function<20.90072148/0 in :erl_eval.expr/5>
iex(one@127.0.0.1)6> Node.spawn(:"two@127.0.0.1", f)
:"two@127.0.0.1"
#PID<9243.75.0>
iex(one@127.0.0.1)7>
```

Can't we do all this with C++/Java

"Any sufficiently complicated concurrent program in another language contains an ad-hoc informally specified bug ridden slow implementation of half of erlang."

[Robert Virding]

Looking ahead...

- Macros
- Protocols
- OTP (Open Telecom Platform)

Learning Resources

- Online: http://elixir-lang.org/getting_started/1.html
- Book: https://pragprog.com/book/elixir/programming-elixir
- Book: http://www.manning.com/juric/
- Online: http://chimera.labs.oreilly.com/books/1234000001642/index.html
- Online: http://learnyousomeerlang.com/ (Erlang)
- Online: http://www.erlang.org/download/armstrong_thesis_2003.pdf