Simpli.fi

A Brief Tour of Elixir

Dan Swain

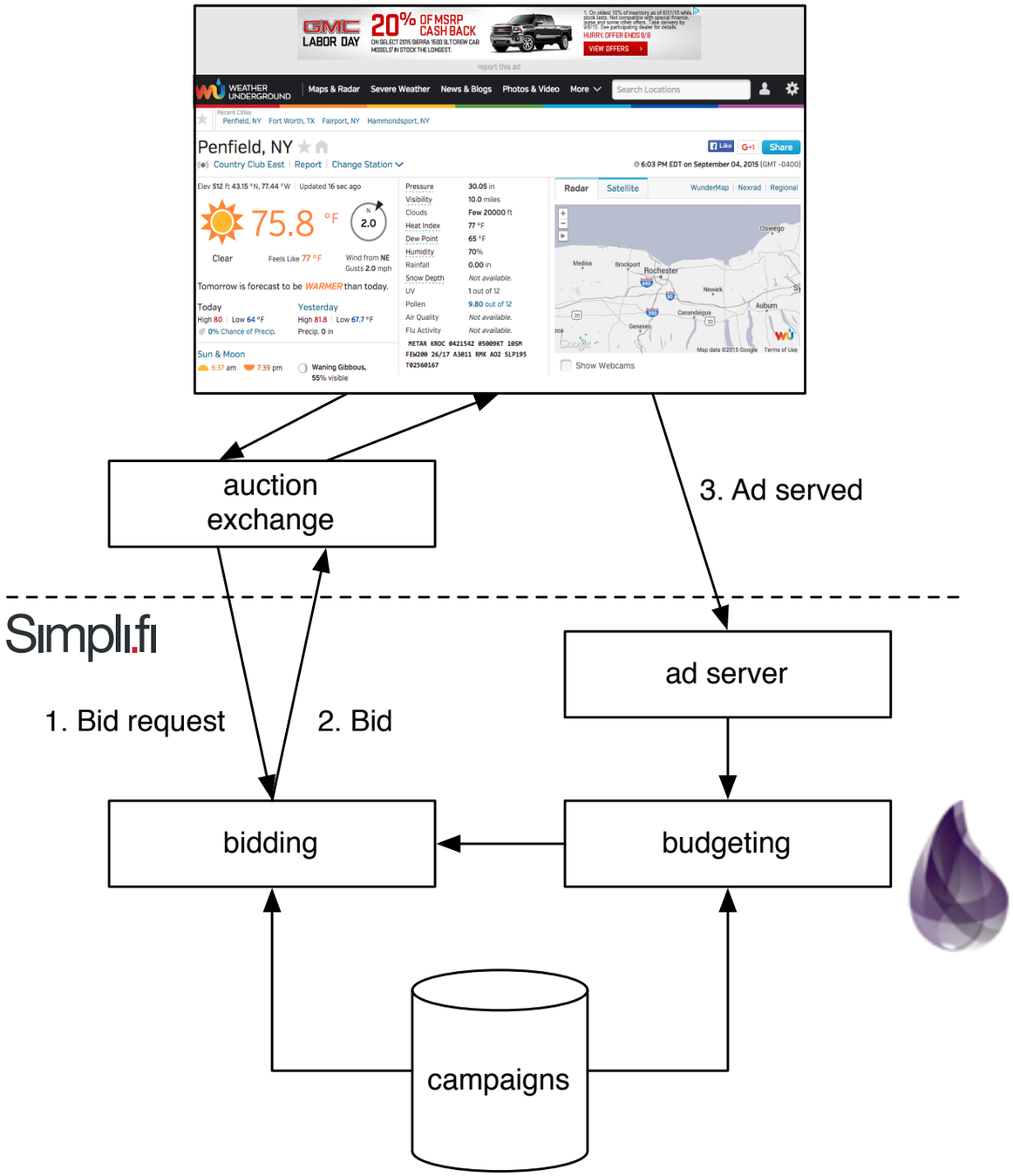
@dantswain | dan.t.swain@gmail.com

Senior Developer, Real-time bidding

simpli.fi

REAL TIME BIDDING

All of this happens while the page loads

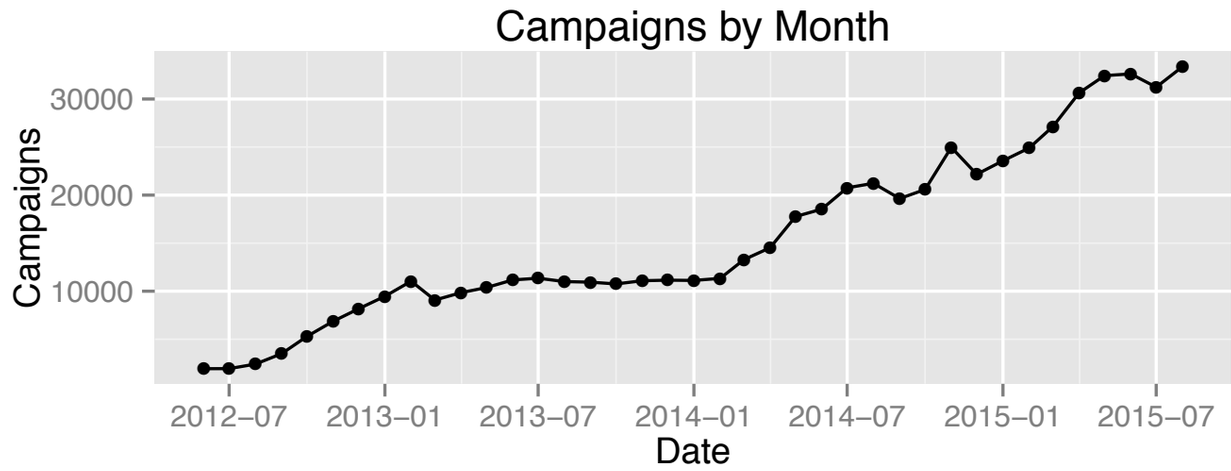


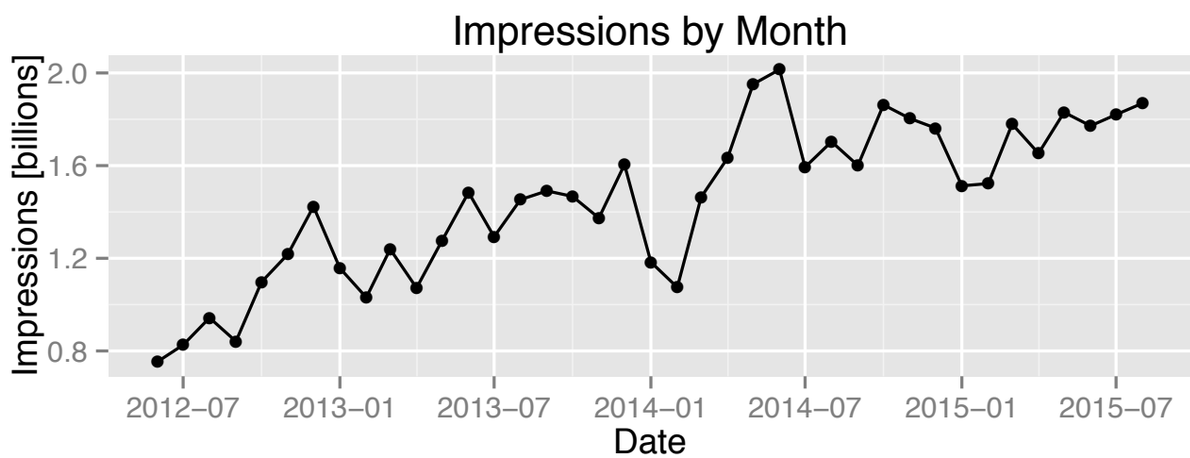
SIMPLI.FI AS A TECH COMPANY

From a developer's perspective

- 1.25M bid requests per second
  — Must be handled in < 10 msec
- 30,000 active ad campaigns
  (186,000 creatives, 10x growth in 3 years)
- 1,000,000,000 user profiles
- 60,000,000 ads served per day
- 5 global data centers
- Constant growth (scaling!)
- Lots of technology
  - C++, Ruby (on and off Rails), Elixir and Erlang, Javascript, LUA, Python....
  - Postgres, Redis, Hadoop, Vertica, Kafka, RabbitMQ

Whatever gets the job done - even if it's something new.





OUTLINE

History

Elixir language basics

OTP and the Real World

HISTORY

- Erlang (1987)
- Elixir (2012)
- My history with the Erlang VM (2012)

A long time ago in a country far, far away...

Erlang - http://erlang.org

1987-1991 - R&D at Ericsson

1991 - First release

As of 2015 - R18

Functional, kind of lisp-ish syntax (lots of commas)

Concurrent & distributed from the ground up

```
-module(sample_erlang).

-export([factorial/1]).

factorial(N) ->
  factorial(N, 1).

factorial(1, Acc) ->
  Acc;
factorial(N, Acc) ->
  factorial(N - 1, N * Acc).
```

Pattern matching

Optimized for tail call recursion

In 2012 in a country far, far away…

Elixir - http://elixir-lang.org

2012 - José Valim, R&D at Platformatec, v0.x released to public

2014 - v1.0 released

As of 1/2016 - v1.2

Functional, Ruby-inspired syntax

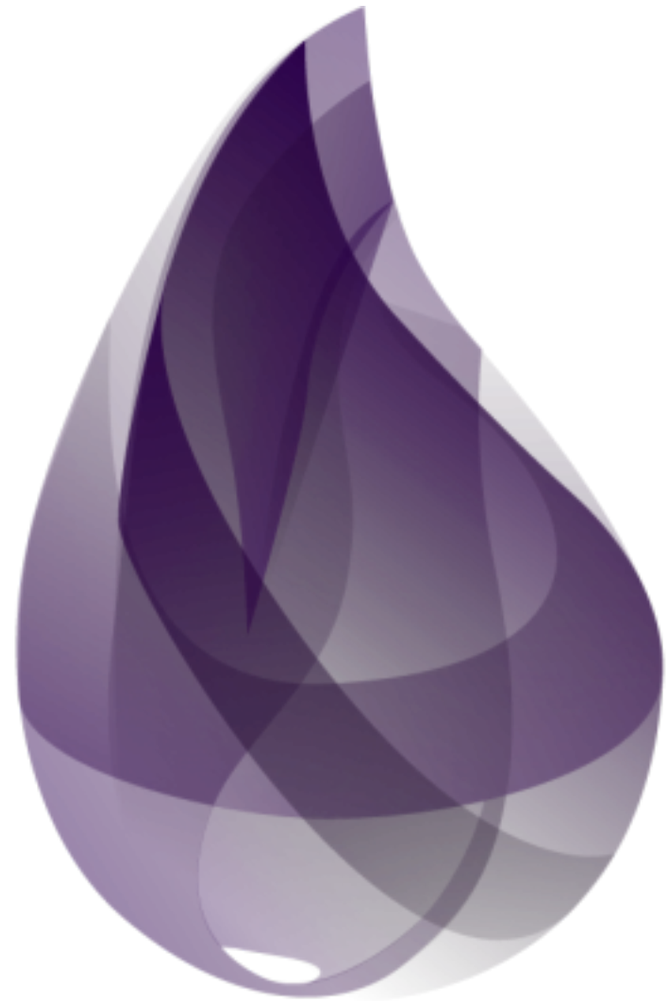
Runs on the Erlang VM with full interop

Designed to be more productive & extensible

```
defmodule SampleElixir do
  def factorial(n) do
    factorial(n, 1)
  end

  defp factorial(1, acc) do
    acc
  end
  defp factorial(n, acc) do
    factorial(n - 1, n * acc)
  end
end
```

Generally less boilerplate



Also in 2012 in a state far, far away…

My history with the Erlang VM

2012 - Rewrite simpli.fi real-time accounting system from Ruby to Erlang

Originally tasked to rewrite in C++

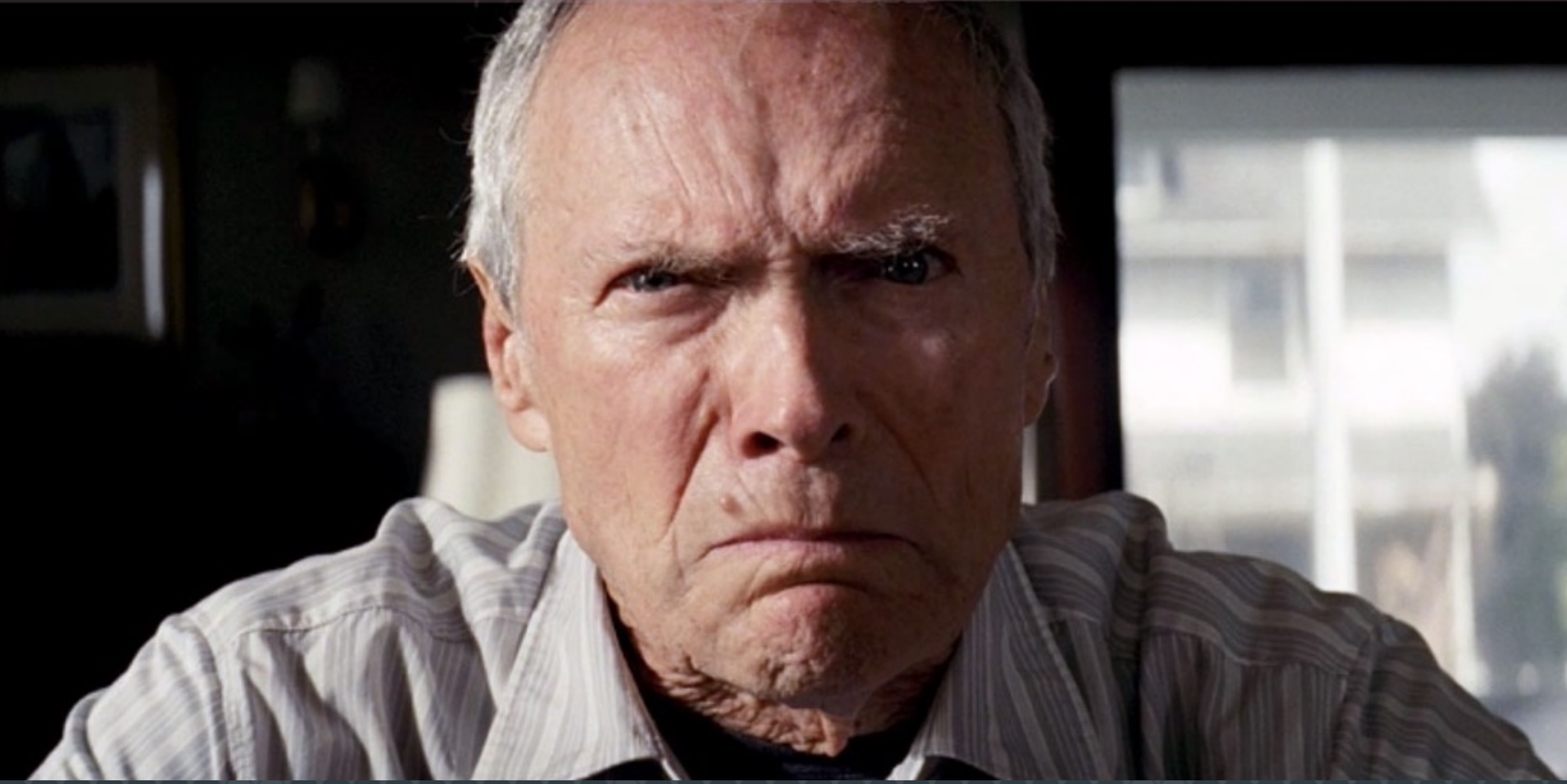
Elixir was brand new, not viable for production

Now in production for over 3 years, very little downtime

Meanwhile... Elixir gaining momentum

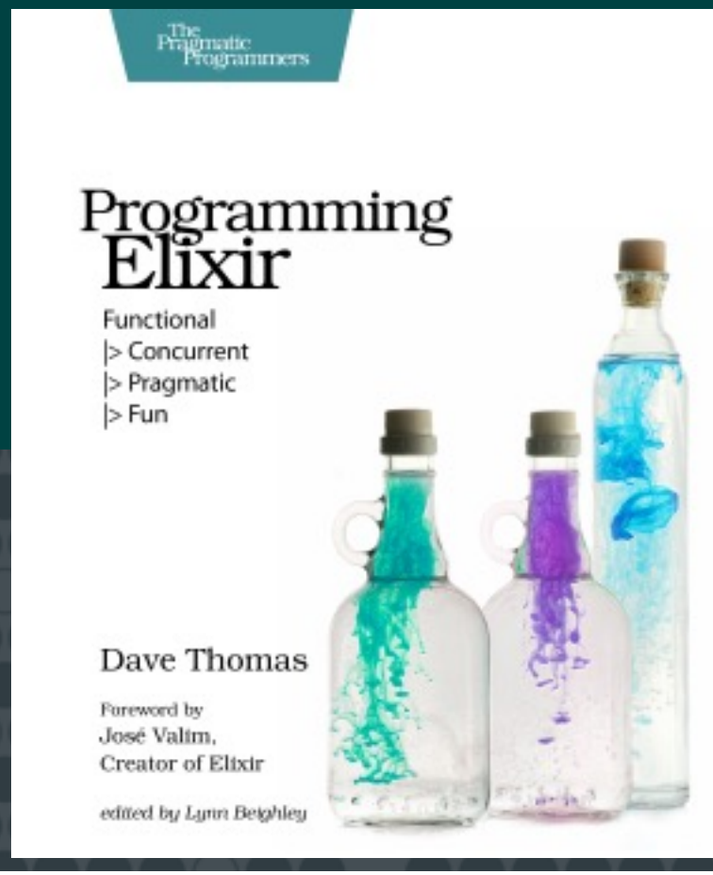
In 2013 in hipster shops everywhere…


ERLANG WAS GREAT
IN THE 90S
memegenerator.net



Erlang not pretty enough for ya?

Well, actually, Elixir has a lot more going on than just the syntax.



Elixir has a lot more going on than just the syntax.

An Elixir convert

2015 - Project converted to Elixir

Build system and tooling

Still some legacy Erlang (full interop!)

All new code in Elixir

Elixir Language Basics

A brief tour

- Single assignment
- Pattern matching
- Anonymous functions & closures
- Pipe operator
- Streams
- Macros
- Type annotation
- Processes

Single assignment

Erlang - true single assignment & immutability

```
Eshell V7.1  (abort with ^G)
1> X = 1.
1
2> X = 2.
** exception error: no match of right hand side value 2
```

Elixir - multiple assignment by default, optional single assignment

```
Interactive Elixir (1.1.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> x = 1
1
iex(2)> x = 2
2
iex(3)> ^x = 3
** (MatchError) no match of right hand side value: 3
```

Pattern matching

```elixir
defmodule PatternMatchingExamples do
  def foo(x) when is_integer(x) do
    IO.puts("#{x} is an integer!")
  end

  def bar(:plus, x, y) do
    x + y
  end
  def bar(:minus, x, y) do
    x - y
  end
  def bar(:times, x, y) do
    x * y
  end
  def bar(:divide, x, y) do
    x / y
  end

  def sum_tuple({x, y, z}) do
    x + y + z
  end
end
```

Anonymous functions, closures, higher order functions

```elixir
defmodule Functions do
  # a closure
  def adder(addend) do
    fn(x) ->
      x + addend
    end
  end

  # taking a function as an argument
  def map_adder(mapper) do
    fn(x1, x2) ->
      mapper.(x1) + mapper.(x2)
    end
  end
end
```

```
iex(2)> add1 = Functions.adder(1)
#Function<1.24638647/1 in Functions.adder/1>
iex(3)> add1.(0)
1
iex(4)> add1.(1)
2
iex(5)> doubler = fn(x) -> 2 * x end
#Function<6.54118792/1 in :erl_eval.expr/5>
iex(6)> double_and_add = Functions.map_adder(doubler)
#Function<0.24638647/2 in Functions.map_adder/1>
iex(7)> double_and_add.(1, 2)
6
iex(8)> Enum.sum(Enum.map([1, 2], fn(x) -> 2 * x end))
6
```

Pipe Operator

Output of previous command => first argument to next command

```elixir
defmodule PipeOperator do
  def without_pipes(x) do
    out1 = Enum.map(x, &double/1)
    sum = Enum.sum(out1)
  end

  def with_pipes(x) do
    x
    |> Enum.map(&double/1)
    |> Enum.sum
  end

  def equivalent_with_pipes(x) do
    x |> Enum.map(&double/1) |> Enum.sum
  end

  defp double(x) do
    2 * x
  end
end
```

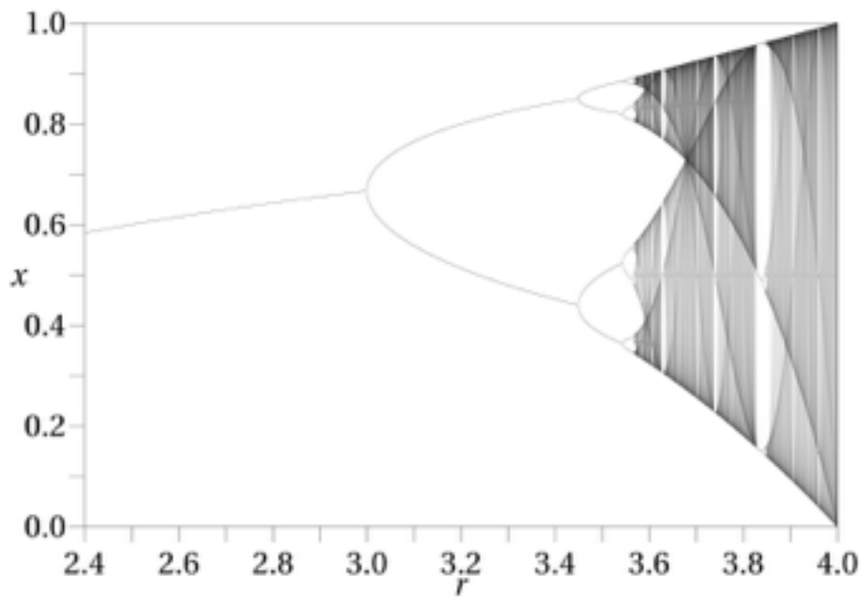
Function reference

Streams

```elixir
defmodule StreamExamples do
  # The logistic stream is chaotic for r between ~ 3.57 and 4
  # see https://en.wikipedia.org/wiki/Logistic_map
  def logistic_stream(r, initial_value) do
    iterator = fn(x) -> r * x * (1 - x) end
    Stream.iterate(initial_value, iterator)
  end

  # Stream a dictionary file, get all words of a certain length
  def load_dictionary(path, word_length) do
    File.stream!(path)
    |> Stream.map(&String.strip/1)
    |> Stream.map(&String.upcase/1)
    |> Stream.filter(fn(word) -> String.length(word) == word_length end)
    |> Enum.to_list
  end
end
```

```
iex(29)> logistic = StreamExamples.logistic_stream(2.5, 0.2)
#Function<29.125745303/2 in Stream.unfold/2>
iex(30)> logistic |> Enum.take(10)
[0.2, 0.4, 0.6, 0.6000000000000001, 0.6, 0.6000000000000001, 0.6,
 0.6000000000000001, 0.6, 0.6000000000000001]
iex(31)> chaotic = StreamExamples.logistic_stream(3.9, 0.2)
#Function<29.125745303/2 in Stream.unfold/2>
iex(32)> chaotic |> Enum.take(10)
[0.2, 0.6240000000000001, 0.9150335999999998, 0.30321373239705673,
 0.8239731430433209, 0.5656614700878645, 0.9581854282490118,
0.1562578420270518, 0.5141811824451928, 0.9742156868513789]
```



https://en.wikipedia.org/wiki/Logistic_map#/media/File:Logistic_Bifurcation_map_High_Resolution.png

Macros

```elixir
defmodule MacroExamples.Macros do
  # define a function for each key in config that
  # returns the corresponding value
  defmacro defconfig(config) do
    config |> Enum.map(fn({k, v}) ->
      quote do
        def unquote(k)() do
          unquote(v)
        end
      end
    end)
  end
end

defmodule MacroExamples do
  import MacroExamples.Macros

  # defines MacroExamples.foo and MacroExamples.bar
  defconfig(foo: 1, bar: 2)
end
```

```
iex(2)> MacroExamples.foo
1
iex(3)> MacroExamples.bar
2
```

The first rule of metaprogramming is:

Avoid metaprogramming.

(If you have to, this is a good reference.)

Type Annotation

```elixir
defmodule TypeExample do
  @type t :: {boolean, non_neg_integer}

  @spec bar(integer) :: t
  def bar(x) when is_integer(x) do
    {x >= 0, abs(x)}
  end

  @spec baz(t) :: non_neg_integer
  def baz({_, x}) do
    x
  end

  def fail1 do
    bar(-1.0)
  end

  def fail2 do
    {1, true} = baz({true, -1})
  end
end
```

```
dswain at itsacomputer in ~/Documents/RocFPElixirTalk
$ elixirc type_example.ex

dswain at itsacomputer in ~/Documents/RocFPElixirTalk
$ dialyzer Elixir.TypeExample.beam
  Checking whether the PLT /Users/dswain/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
type_example.ex:4: Function fail1/0 has no local return
type_example.ex:5: The call 'Elixir.TypeExample':bar(float()) will never return since the success typing is (integer())
-> {'false',non_neg_integer()} | {'true',non_neg_integer()} and the contract is (integer()) -> t()
type_example.ex:8: Function fail2/0 has no local return
type_example.ex:9: The call 'Elixir.TypeExample':baz({'true',-1}) breaks the contract (t()) -> non_neg_integer()
 done in 0m8.73s
done (warnings were emitted)
```

Processes

- Lightweight, not the same as linux processes
- SMP - Takes advantage of multi-processor machines automatically

`spawn(function)` - Spawn a process that executes `function`, returns its pid

```
iex(9)> spawn(fn() -> :timer.sleep(10000); IO.puts("HI 1") end)
#PID<0.1781.3>
iex(10)> spawn(fn() -> :timer.sleep(3000); IO.puts("HI 2") end)
#PID<0.1783.3>
HI 2
HI 1
```

`send(pid, message) & receive` - Communication between processes

Processes

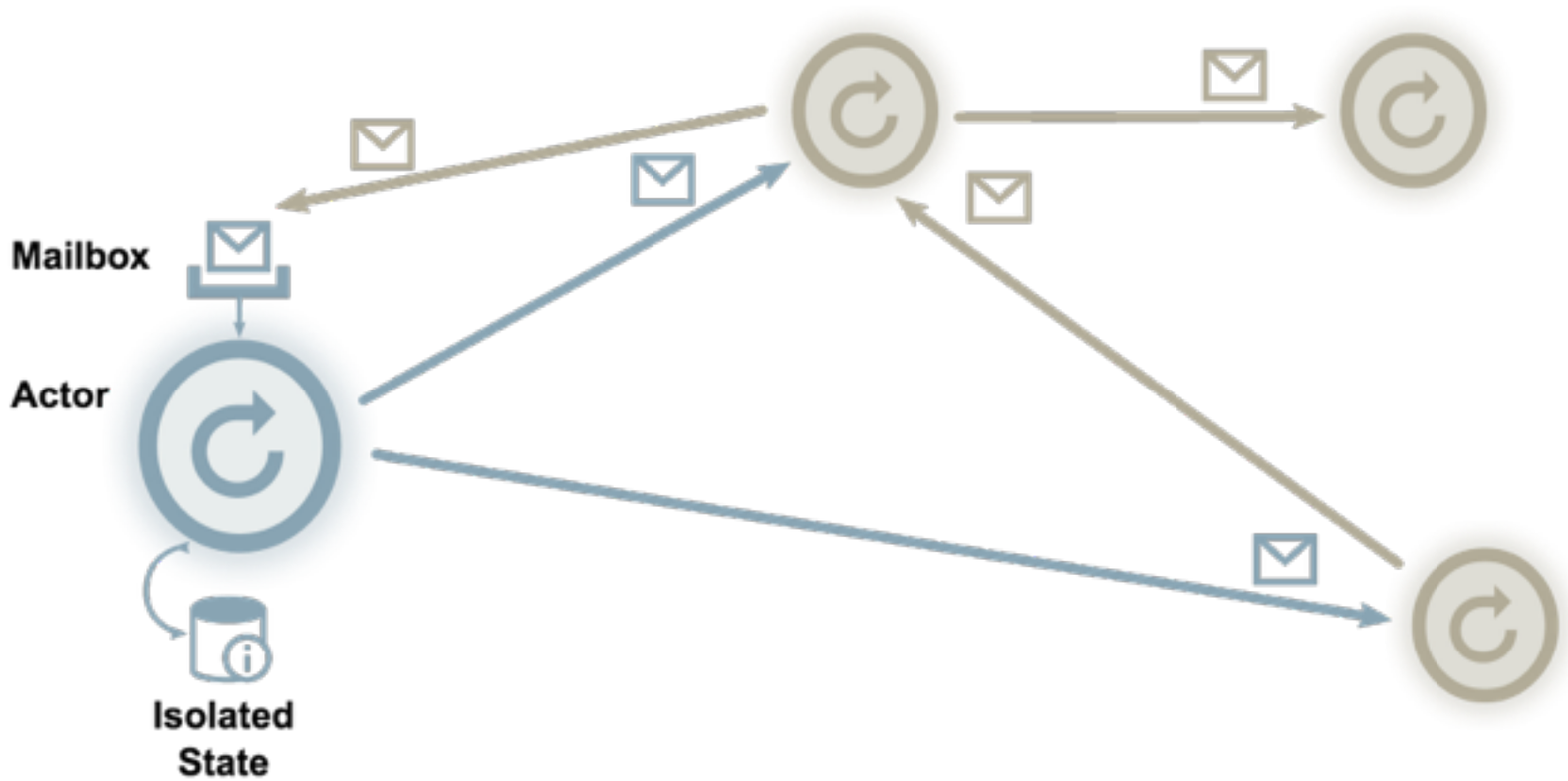
```elixir
defmodule EchoProc do
  # spawns a new process and returns its pid
  def new(name, attention_span) do
    spawn(fn() -> loop(name, attention_span) end)
  end

  # simple send/receive API
  def speak_at(pid, x) do
    # NOTE whatever process calls this function will be 'self()'
    # e.g., our console
    send(pid, {x, self()})
    receive do x -> x end
  end

  defp loop(name, attention_span) do
    receive do
      {:exit, _from} ->
        IO.puts("#{name} is shutting down")
        # no recursive call -> process finishes
      {x, from} ->
        IO.puts("#{name} got #{inspect x}")
        send(from, String.reverse(inspect x))
        loop(name, attention_span)
    after
      attention_span ->
        IO.puts("#{name}: I AM BORED")
        loop(name, attention_span)
    end
  end
end
```

Processes

- Processes are the key to solving problems in Erlang/Elixir
- Actor model - https://en.wikipedia.org/wiki/Actor_model
- Processes need not be *on the same hardware*
  - Built-in distributability



http://berb.github.io/diploma-thesis/original/054_actors.html

INTERLUDE:

Your first production experience with the Erlang VM


PROCESSES
PROCESSES EVERYWHERE
ator.net


HyFlex
AON
LUKOIL
CHEVROLET
CRUZE
KW

Oh @#%@#$!


SCOTT
DARE
STUNT

Really more like


Bensalem
EXIT 351
EXIT 352

OTP and the Real World

Building and deploying Elixir applications

- GenServer
- OTP Applications & Supervision Trees
- Tooling
- Production - Attaching to a live REPL
- Distributed systems
- Phoenix

GenServer

Remember this guy?

```
defmodule EchoProc do
  # spawns a new process and returns its pid
  def new(name, attention_span) do
    spawn(fn() -> loop(name, attention_span) end)
  end

  # simple send/receive API
  def speak_at(pid, x) do
    # NOTE whatever process calls this function will be 'self()'
    # e.g., our console
    send(pid, {x, self()})
    receive do x -> x end
  end

  defp loop(name, attention_span) do
    receive do
      {:exit, _from} ->
        IO.puts("#{name} is shutting down")
        # no recursive call -> process finishes
      {x, from} ->
        IO.puts("#{name} got #{inspect x}")
        send(from, String.reverse(inspect x))
        loop(name, attention_span)
    after
      attention_span ->
        IO.puts("#{name}: I AM BORED")
        loop(name, attention_span)
    end
  end
end
```

GenServer

The “right” way

```elixir
defmodule EchoServer do
  use GenServer

  defmodule State do
    defstruct name: nil, attention_span: nil
  end

  # API

  def start_link(name, attention_span) do
    GenServer.start_link(__MODULE__, {name, attention_span})
  end

  def stop(pid) do
    GenServer.call(pid, :stop)
  end

  def speak_at(pid, x) do
    GenServer.call(pid, {:spoken_at, x})
  end

  # GenServer callbacks

  def init({name, attention_span}) do
    {:ok,
     %State{name: name, attention_span: attention_span},
     attention_span}   # timeout
  end

  def handle_call(:stop, _from, state) do
    {:stop, :normal, :ok, state}
  end
  def handle_call({:spoken_at, x}, _from, state) do
    IO.puts("#{state.name} got #{inspect x}")
    {:reply, String.reverse(inspect x), state, state.attention_span}
  end

  def handle_info(:timeout, state) do
    IO.puts("#{state.name}: I AM BORED")
    {:noreply, state, state.attention_span}
  end

  def terminate(reason, state) do
    IO.puts("#{state.name} is shutting down because #{inspect reason}")
    :ok
  end
end
```

OTP, the Open Telecom Platform

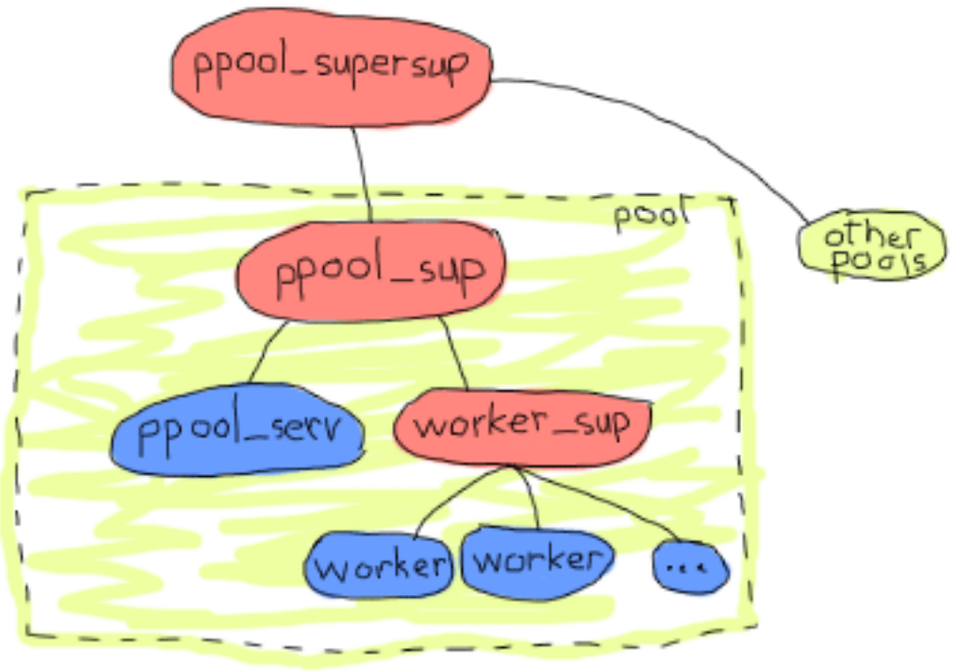
- From Erlang / Ericsson, really not about Telecom at all
- OTP Application ~~ Dependency unit
- Any number of OTP applications per VM

Supervision Trees

- Automatic restart, “fail fast” approach
- Linked processes, e.g., parent fails => child fails
- One goal is to not worry about state

In Elixir

- Baked into the tooling & documentation
- Much less boilerplate than Erlang



http://learnyousomeerlang.com/static/img/ppool_supersup.png

Tooling in Elixir

- mix = build tool, dependency manager, task runner, etc…
  - mix new PATH [--sup] [--module MODULE] [--app APP] [—umbrella]
  - mix.exs = Project config
  - mix compile, mix test, mix dialyze, mix deps, mix run, etc…
  - MIX_ENV => production / test / development (config files)
- hex = package manager
  - http://hex.pm - community repository
  - Can do local deps, git/github, etc.
- ExUnit = test framework
  - Setup/teardown, Applications, Supervision trees, etc.
  - Doctesting = awesome

Production

- exrm = deployment packager - https://github.com/bitwalker/exrm
- Attaching a REPL to your running app is ridiculously useful
- RPC against running app (e.g., cron / monitoring)
- Live upgrades are theoretically possible
    - But almost never worth it in my experience
- Supervision trees == super reliable

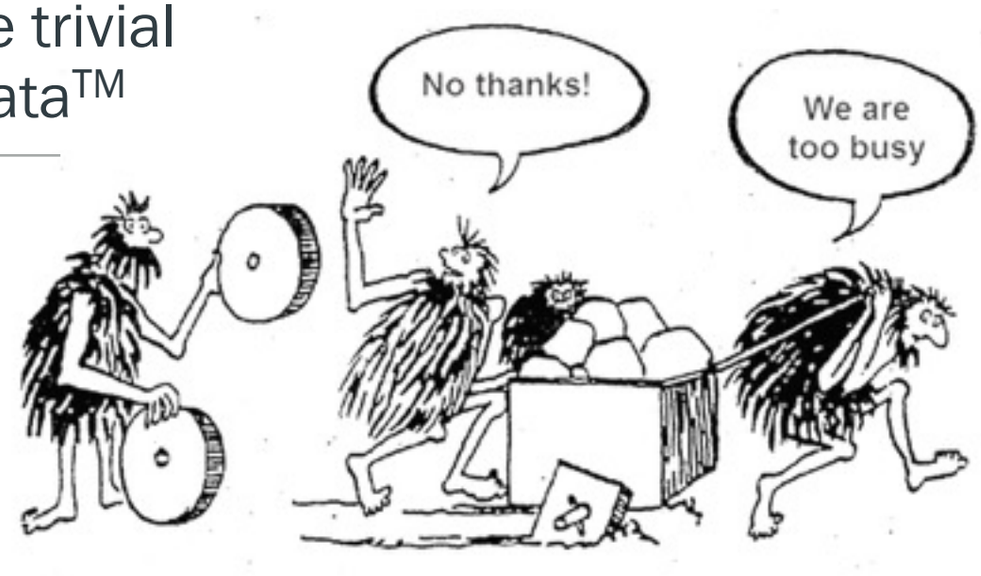
Distributed systems: Don't even.

Yes, the Erlang VM is great for building distributed systems

No, that does not mean you should expect it to be trivial to fire up a hundred servers to crunch your Big Data™

There ARE good use cases for rolling your own distributed system using the Erlang VM

- Riak
- RabbitMQ
- ejabberd
- ~~Your application~~



RabbitMQ is a great message broker system built in Erlang

- Rich set of messaging primitives
- Great official Erlang client - messages delivered via gen_server
- Separate your business logic from your infrastructure

You have plenty of work to do figuring out how to solve your own problems

Distributed systems: OK, fine.

Tutorial: http://elixir-lang.org/getting-started/mix-otp/distributed-tasks-and-configuration.html

Node bar - has module “Hello” loaded

```
iex(bar@itsacomputer)1> c("hello.ex")
[Hello]
iex(bar@itsacomputer)2> Hello.world
Hello, World!
:ok
```

Node foo - does NOT have module “Hello” loaded

```
iex(foo@itsacomputer)1> Hello.world
** (UndefinedFunctionError) undefined function: Hello.world/0 (module Hello is not available)
    Hello.world()
iex(foo@itsacomputer)1> Node.spawn_link(:"bar@itsacomputer", fn -> Hello.world end)
Hello, World!
#PID<9279.76.0>
```

Nodes foo and bar need not be on the same hardware!

Phoenix

- Full-featured web framework
- Relatively mature now (post 1.0 as of 8/2015)
- Very fast
- Concurrent by nature
- Web sockets are first-class citizens
- Inspired by Rails, but with a healthy dose of caution re: metaprogramming
- Ecto - “O”RM, functional, composable queries

http://www.phoenixframework.org/

TAKEAWAYS

Elixir is...

- Functional
- Fundamentally concurrent
- A modern language with useful tooling and documentation
- Rock solid in production
- Distributable
- A joy to work with

THANK YOU!

@dantswain | github.com/dantswain | dantswain.com

Simpli.fi

(Is hiring! jobs@simpli.fi)

BACKUP SLIDES