



elixir and OTP apps

introduction

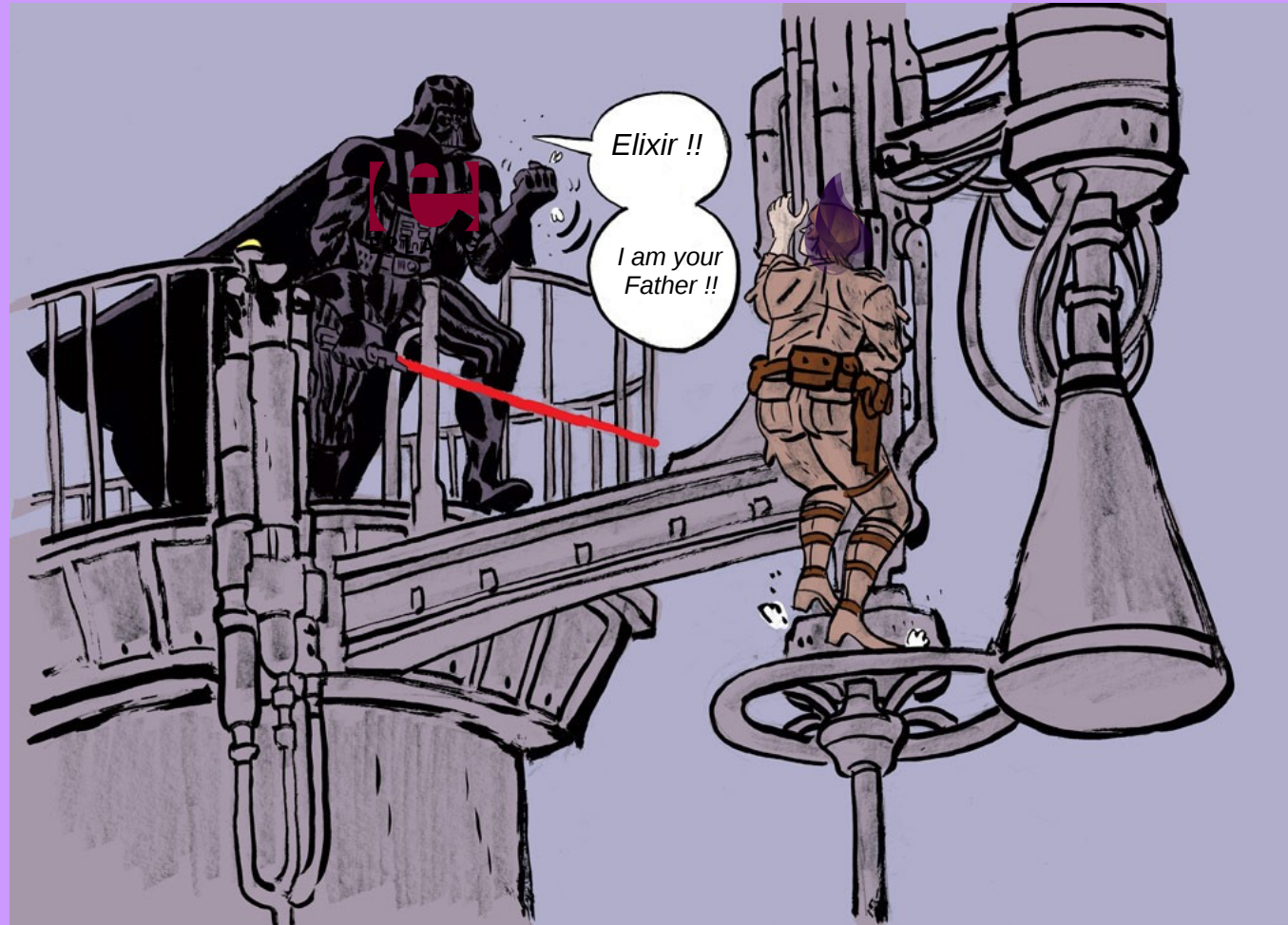
content

- Elixir
 - history
 - iex
 - types
 - operators
 - patter matching
 - control flow structures
 - functions
 - modules
 - recursion
 - pipe operator
 - comprehensions
 - processeses
- OTP
 - scalability
 - fault tolerance
 - concurrency
 - history
 - behaviours
 - ets
 - mix
- Workshop
 - observer
- Resources
- Q&A

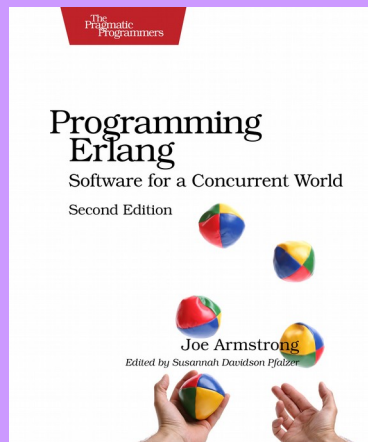


elixir

history

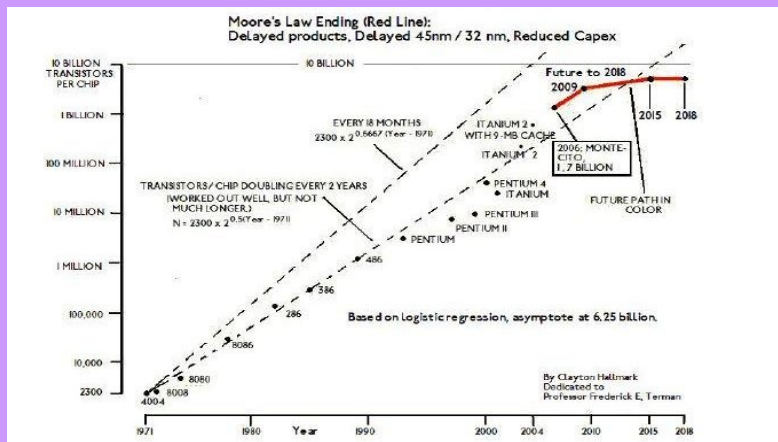


history



- Joe designed erlang in the 80s, when he was an Ericsson engineer, focusing on:
- Scalability
 - Fault Tolerance
 - Concurrency

history



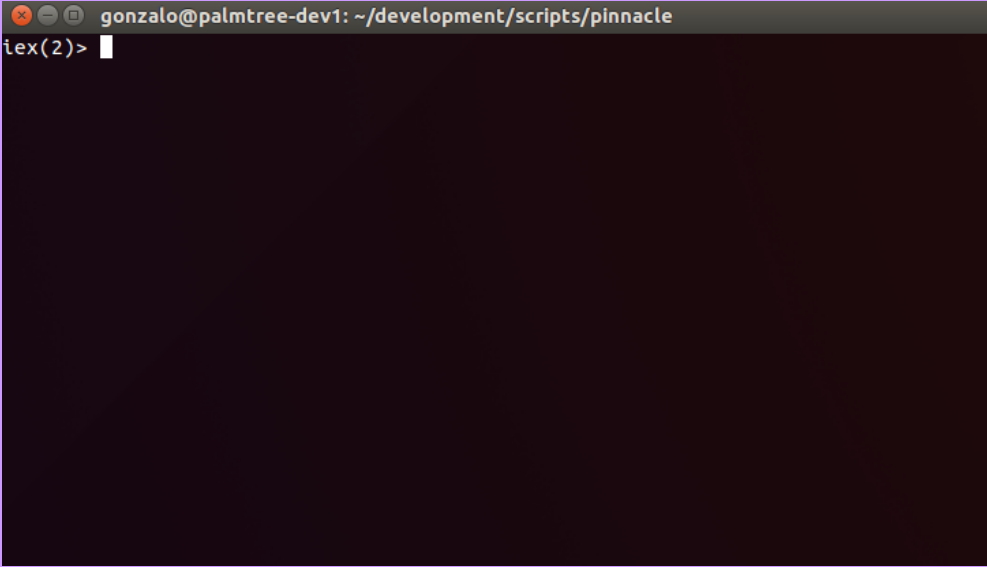
- Moore's law is almost over, but it is too easy to have multicore pc.
- Languages like Python, Javascript or Ruby can't take advantage of several CPUs

history



Jose Valim tried to improve Ruby on Rails framework, but he could not. Then, he discovered how Erlang worked, but he did not like the syntax, therefore he decided to create a new language which will run in the BEAM (Erlang Virtual Machine).

iex

A terminal window with a dark background and light gray text. The title bar at the top shows a red close button, a yellow minimize button, and a green maximize button, followed by the text "gonzalo@palmtree-dev1: ~/development/scripts/pinnacle". The main area of the terminal displays the prompt "iex(2)>" followed by a white cursor bar.

```
gonzalo@palmtree-dev1: ~/development/scripts/pinnacle  
iex(2)> 
```


types

Elixir is a dynamic language, but of course it has types:

1	INTEGER
0X1F	INTEGER
1.0	FLOAT
true	BOOLEAN*
:atom	ATOM
"string"	STRING
[1, 2, 3]	LIST
{1, 2, 3}	TUPLE
%{id: 1, text: "example"}	MAP
%{"id" => 1, "text": "example"}	MAP
pid(23, 32, 45)	PID
nil	*

operators

+	$2 + 2 = 4$
-	$3 - 2 = 1$
*	$2 * 2 = 4$
/	$10 / 2 = 5$
++	$[1, 2, 3] ++ [4, 5, 6] = [1, 2, 3, 4, 5, 6]$
<>	"foo" <> "bar" = "foobar"
and	true and false = false
or	true or false = true
!	!true = false
	$1 \text{false} = 1$
&&	nil && 13
==	$1 == 1 = \text{true}$
!=	$1 != 1 = \text{false}$
===	$1 === 1.0 = \text{false}$
<	$1 > 1 = \text{false}$
>	$1 < 1 = \text{false}$

pattern matching

The pattern match operator is “=”

```
iex()> x = 1
1
iex()> {a, b, _} = {:ok, 2, 3}
{:ok, 2, 3}
iex()> [head | tail] = [1, 2, 3]
[1, 2, 3]
iex()> head
1
iex()> tail
[2, 3]
iex()> %{id: id} = %{id: 1, text: "example"}
%{id: 1, text: "example"}
iex()> id
1
```

control flow structures

Case compares a value against many patterns until we find a matching one:

```
iex> x = {1, 2, 3}
{1, 2, 3}
iex> case x do
...>   {2, 3, 4} → "First option"
...>   {1, _, 3} → "Second Option"
...>   _ → "It always matches"
...> end
"Second Option"

iex> x = {1, 7, 3}
{1, 7, 3}
iex()> case x do
...>   {2, 3, 4} → "First option"
...>   {1, _, 3} → "Second Option"
...>   _ → "It always matches"
...> end
"Second Option"
```

You can also use guards in your pattern matching: `{1, z, 3} when z > 0 → "guards !!!"`

control flow structures

Cond allows us to check different conditions until we find the first true.

```
iex> x = {1, 2, 3}
{1, 2, 3}
iex> cond do
...>   x == {2, 3, 4} → "First option"
...>   is_list(x) → "It is a list !!"
...>   2 + 5 == x → " it is 7 !!"
...>   true → "It always matches"
...> end
"It always matches"
```

control flow structures

If and **unless** are macros, but you can normally use them ;).

```
iex> x = {1, 2, 3}
{1, 2, 3}
iex> if {1, 2, 3} == x do
...>   "Yes, it is"
...> else
...>   "No, it is not"
...> end
"Yes, it is"

iex> unless {1, 2, 3} == x do
...>   "No, it is not"
...> else
...>   "Yes, it is"
...> end
"Yes, it is"
```


functions

Functions are first citizens in elixir. Don't forget the "." if it is an anonymous one.

```
iex> square = fn x → x*x end  
Function<6.52032458/1 in :erl_eval.expr/5>  
iex> square.(2)  
4
```

```
iex>my_very_difficult_algo = fn x -> square.(x) end  
#Function<6.52032458/1 in :erl_eval.expr/5>  
iex> my_very_difficult_algo.(2)  
4
```

modules

The functions are grouped in **Modules**.

```
iex> defmodule MyCalculator do
...>   def square(x) do
...>     x * x
...>   end
...> end
{:module, MyCalculator,
 <<70, 79, 82, 49, 0, 0, 4, 212, 66, 69, 65, 77, 69,
 120, 68, 99, 0, 0, 0, 147,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105,
 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 4, 104, 2, ...>>, {:square,
 1}}
```



```
iex> MyCalculator.square(2)
4
```

modules

Elixir has a lot of standard modules that you will take advantage of them.

- Enum
- List
- Map
- Stream
- String
- Integer
- Process
- ...

modules

But Elixir works in the BEAM, so we can also use all the standard modules of Erlang.

- `:os`
- `:calendar`
- `:rand`
- `:ets`
- `:crypto`
- `:global`
- ...

recursion

How are we going to code my things if I don't have loops?

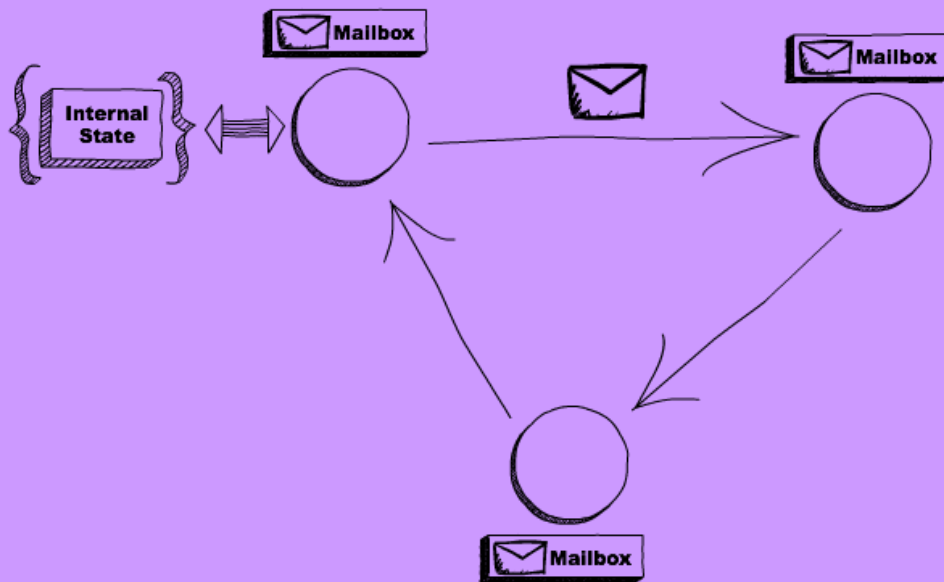
```
iex> defmodule Fib do
...>   def calculate(x), do: fib(x)
...>   defp fib(1), do: 1
...>   defp fib(2), do: 1
...>   defp fib(x), do: fib(x - 1) + fib(x - 2)
...> end
{:module, Fib,
 <<70, 79, 82, 49, 0, 0, 6, 20, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 163,
 131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99,
 115,
 95, 118, 49, 108, 0, 0, 0, 4, 104, 2, ...>>, {:fib, 4}}
```



```
iex> Fib.calculate(7)
13
```

Processes

Elixir (and Erlang) follows the actor model paradigm.



- Every process (Erlang process) has his own mailbox and it communicates using messages

Processes

Self communication !!

```
iex> Process.send(self(), "Hello from Process.send()", [])
iex> send(self(), "Hello send()")
iex> receive do
...>  msg → msg
...> end
"Hello from Process.send()"

iex> receive do
...>  msg → msg
...> end
"Hello from send()"
```

Processes

Communication from outside !!

```
iex> spawn(fn -> send(parent, {:hello, self()})) end)
#PID<0.48.0>
iex> receive do
...>   {:hello, pid} -> "Got hello from #{inspect pid}"
...> end
Got hello from #PID<0.48.0>
```

Processes

I highly recommend to use Task !!

```
iex> Task.start(fn -> send(parent, {:hello, self()}) end)  
{:ok, #PID<0.48.0>}
```

```
iex> receive do
```

```
...>  {:hello, pid} -> "Got hello from #{inspect pid}"
```

```
...> end
```

```
Got hello from #PID<0.48.0>
```

```
iex> task = Task.async(fn -> 2 + 2 end)
```

```
%Task{owner: #PID<0.80.0>, pid: #PID<0.82.0>, ref:  
#Reference<0.0.1.211>}
```

```
iex> middle_sum = 3 + 5
```

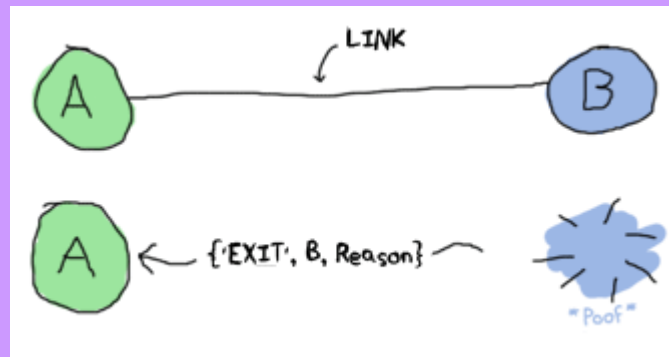
```
8
```

```
iex> result = middle_sum + Task.await(task)
```

```
12
```

processes

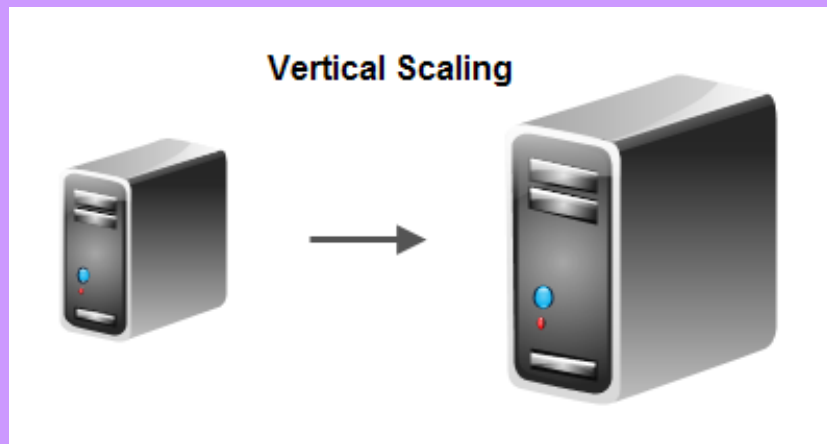
- Linked Processes: If one process dies, the other dies as well
- Monitored Process: If a monitored process dies, the monitor process will receive a notification.



- Linking: You can start a process with the function `start_link()` instead of `start` or just use `Process.link(pid)` from one process indicating the pid of the other process.
- Monitoring: Execute `Process.monitor(pid)` from one process indicating the pid of the other process.



scalability

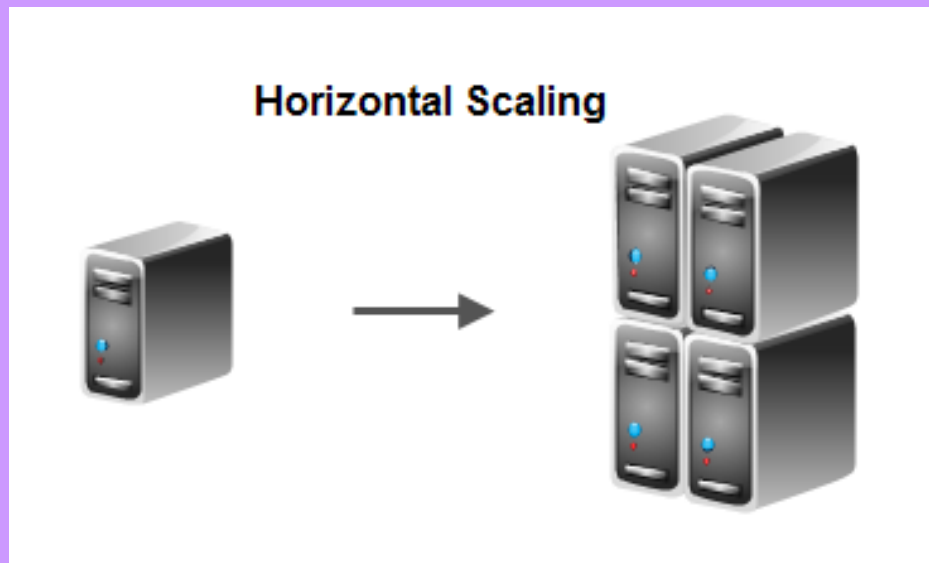


Vertical scalability means: “If you have a ‘bigger’ server, your SW will take advantage of it.”



We have already talk about vertical scalability, but I love this gift

scalability

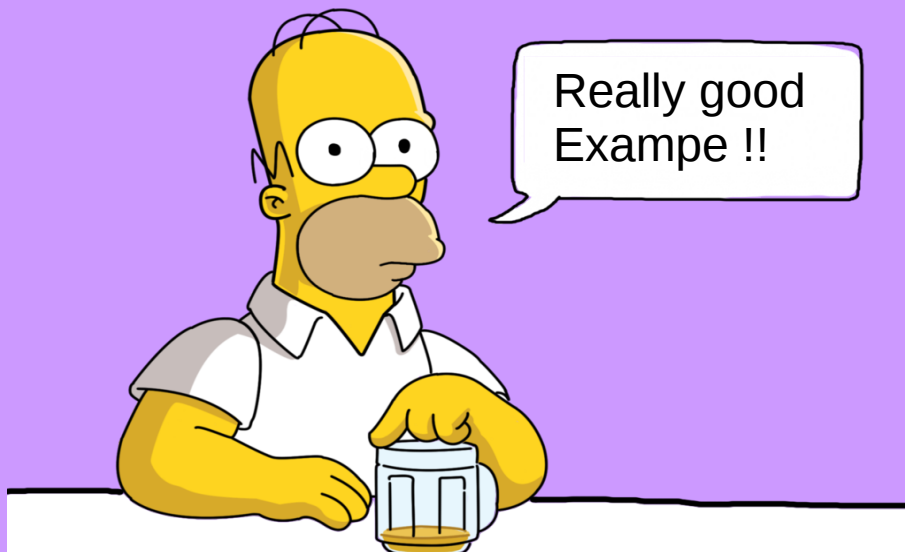


Vertical scalability can reach some limitations, but we need to scale, the solution can be horizontal scalability, or adding more servers. It also implies distribution.

fault tolerance

The only SW that has no bugs is the SW that has never been executed.

SWs which run indefinitely have to be fault tolerance.



Ariane 5 rocket



The rocket self-destructed 37 seconds after launch

Reason: A control software bug that went undetected

Conversion from 64-bit floating point to 16-bit signed integer value had caused an **exception**

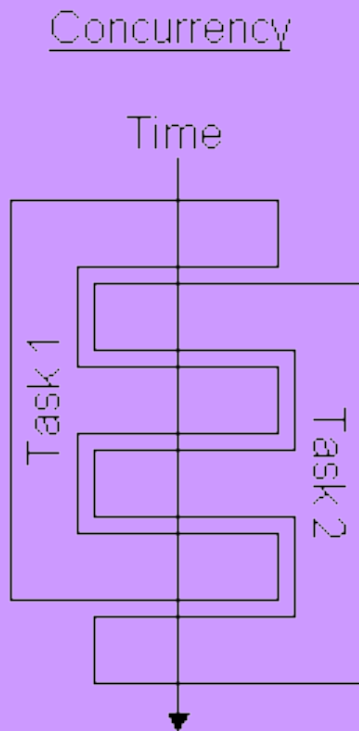
The floating point number was larger than 32767 (max 16-bit signed integer)

Efficiency considerations had led to the disabling of the exception handler.

Program crashed → rocket crashed

Total Cost: over \$1 billion

concurrency



Although we are able to scale in order to use more CPU cores, sure that you will have more tasks than cores, so you need concurrency.

history



Ericsson OTP Team

Twitter: [@erlang_org](https://twitter.com/erlang_org)

Having all these things in mind, the Ericsson guys developed OTP which is a collection of useful middle-ware, libraries, and tools written in Erlang programming language.

behaviours

Behaviours in Elixir (and Erlang) are a way to separate and abstract the generic part of a component (which becomes the behaviour module) from the specific part (which becomes the callback module).

The following are provided by OTP (and some new ones from Elixir):

- Supervisor
- GenServer
- GenStage
- Agent
- GenEvent

behaviours

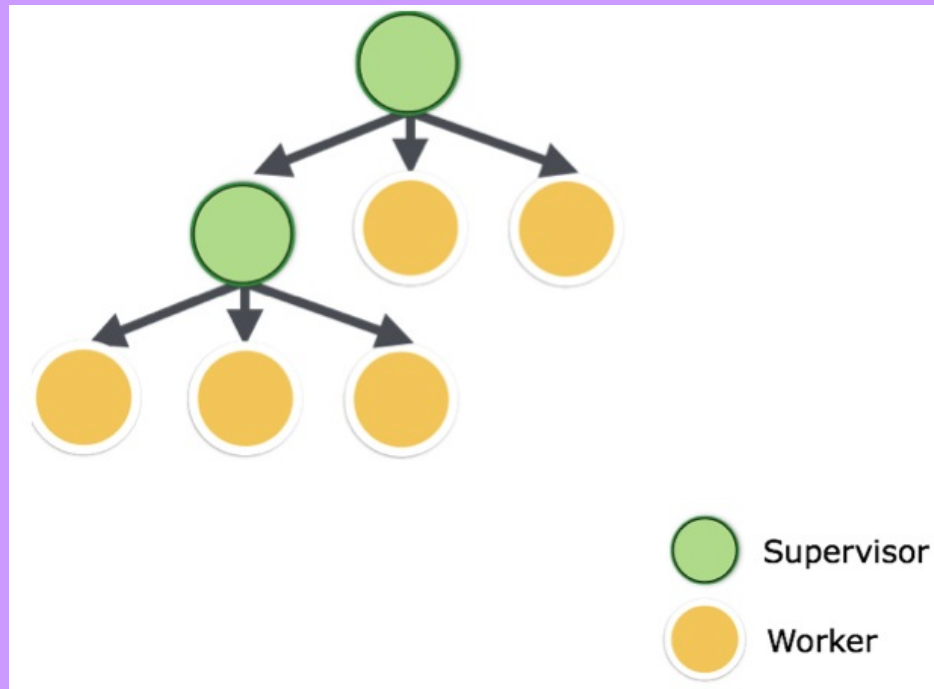
A **Supervisor** is a process which supervises other processes, which are referred to as child processes. (Do you remember the linking/monitoring actions?)

```
# Import helpers for defining supervisors
import Supervisor.Spec

# Supervise the Stack server which will be started with
# a single argument [:hello] and the default registered
# name of MyStack.
children = [
  worker(MyProcess, [[:hello], [name: :my_process]])
]

# Start the supervisor with our child
{:ok, pid} = Supervisor.start_link(children, strategy:
:one_for_one)
```


behaviours



A Supervisor can supervise a worker (Elixir Process) or other supervisors.

behaviours

A **GenServer** is a process like any other Elixir process and it can be used to keep state, execute code asynchronously and other interesting stuffs providing a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into a supervision tree.

```
defmodule Stack do
  use GenServer

  # Callbacks

  def handle_call(:pop, _from, [h | t]) do
    {:reply, h, t}
  end

  def handle_cast({:push, item}, state) do
    {:noreply, [item | state]}
  end
end

# Start the server
{:ok, pid} = GenServer.start_link(Stack, [:hello])

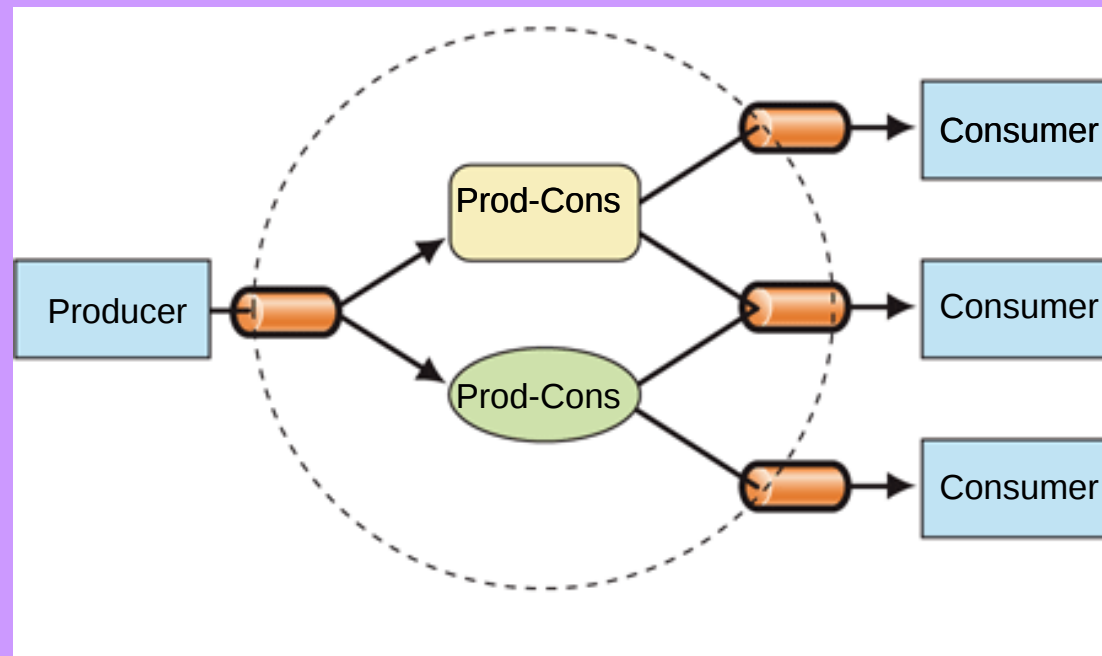
# This is the client
GenServer.call(pid, :pop)
#=> :hello

GenServer.cast(pid, {:push, :world})
#=> :ok

GenServer.call(pid, :pop)
#=> :world
```

behaviours

- GenStage provides the interface to create a distributed set of producer, producer-consumer and consumer processes with back-pressure mechanism. It is based on GenServer.



ETS is a cache system which allows us to store any Elixir term in an in-memory table.

```
iex> table = :ets.new(:my_table, [:named_table])  
:my_table
```

```
iex> :ets.insert(table, {"foo", self})  
true
```

```
iex> :ets.lookup(table, "foo")  
[{"foo", #PID<0.80.0>}]
```

```
iex> :ets.delete(table, "foo")  
true
```

Mix is a build tool that ships with Elixir that provides tasks for creating, compiling, testing your application, managing its dependencies and much more

```
$> mix new my_project --sup
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/my_project.ex
* creating test
* creating test/test_helper.exs
* creating test/my_project_test.exs
```

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

```
cd my_project
mix test
```

Run "mix help" for more commands.

```
$> cat my_project/mix.exs
defmodule MyProject.Mixfile do
  use Mix.Project

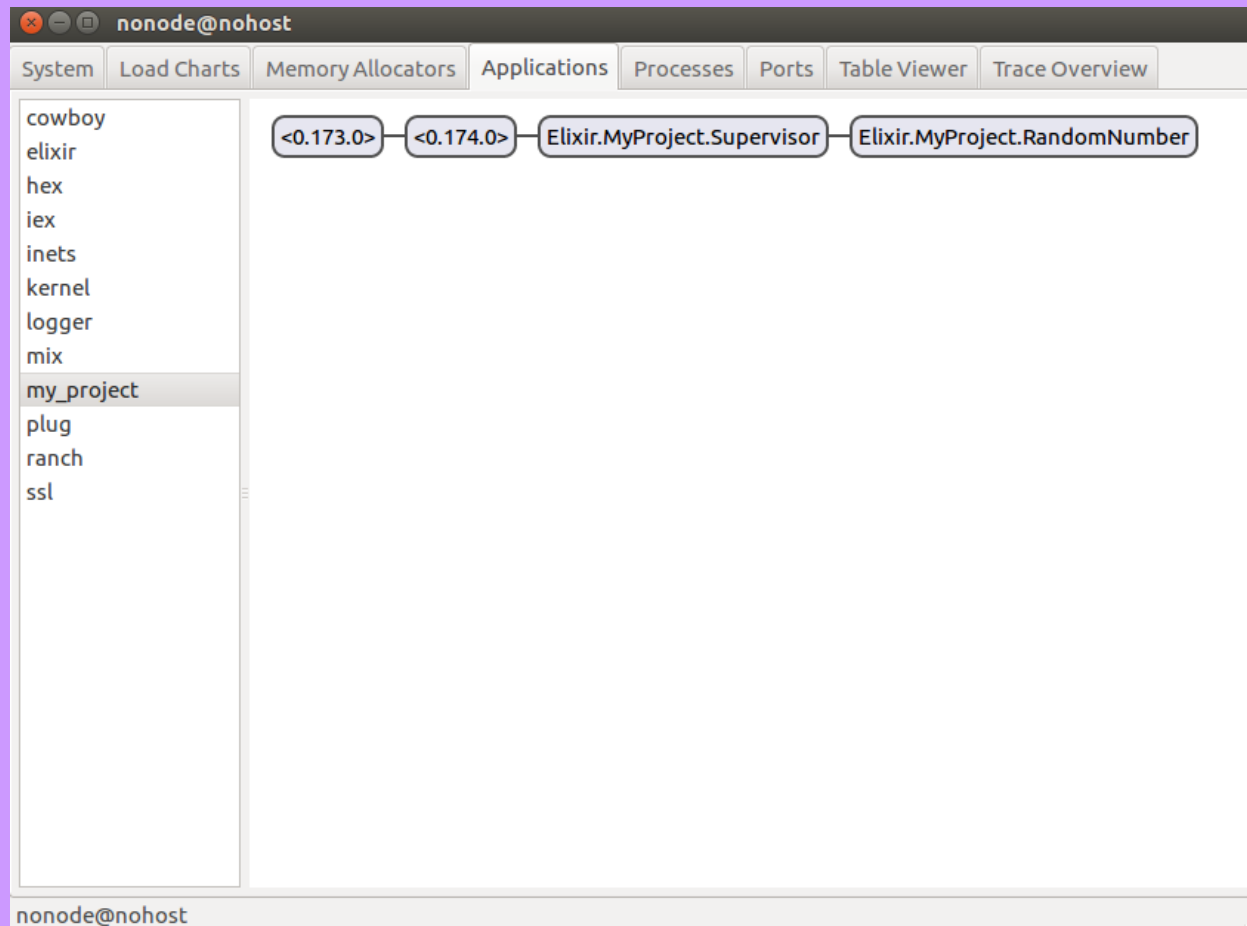
  def project do
    [app: :my_project,
     version: "0.1.0",
     elixir: "~> 1.3",
     build_embedded: Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps: deps()]
  end

  # Configuration for the OTP application
  #
  # Type "mix help compile.app" for more information
  def application do
    [applications: [:logger],
     mod: {MyProject, []}]
  end

  # Dependencies can be Hex packages:
  #
  # {mydep, "~> 0.3.0"}
  #
  # Or git/path repositories:
  #
  # {mydep, git: "https://github.com/elixir-lang/mydep.git", tag: "0.1.0"}
  #
  # Type "mix help deps" for more examples and options
  defp deps do
    []
  end
end
```

Workshop

observer

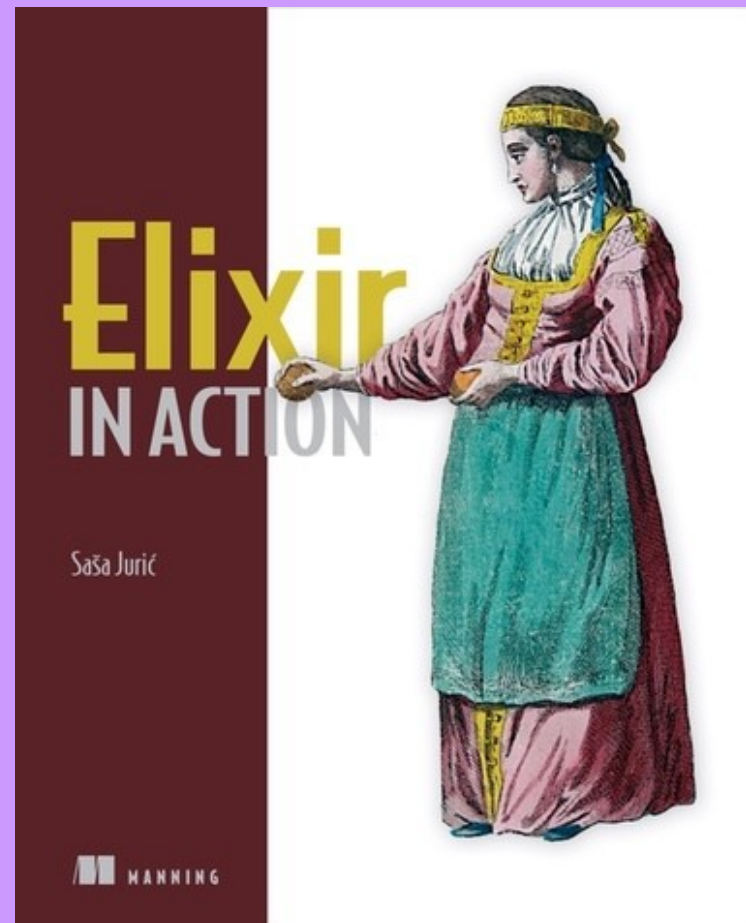
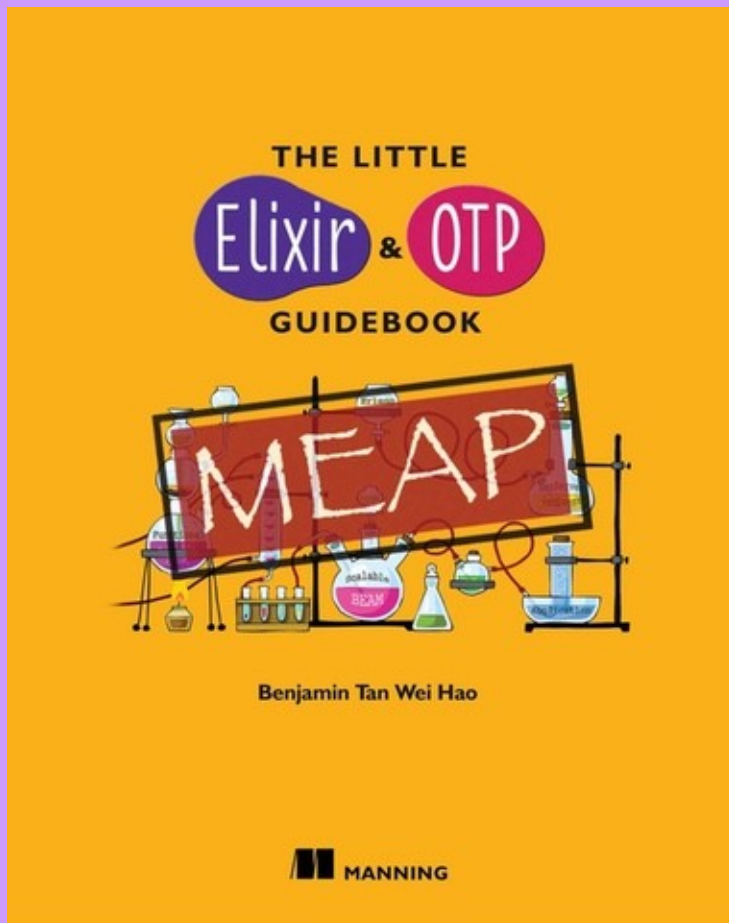


Resources

resources

- <http://elixir-lang.org/>
- <http://elixir-lang.org/docs/stable/elixir/>
- http://erlang.org/doc/reference_manual/
- <https://github.com/elixir-lang/elixir>
- <https://elixir-slackin.herokuapp.com/>
- <https://www.elixirforum.com/>

resources



Q&A