Software Technology of Internet of Things

Drinking the Elixir

Aslak Johansen asjo@mmmi.sdu.dk

Feb 11, 2025

Part 1:
Introduction

Erlang

Back in the 80's the only place where programmers created distributed systems were telephone exchange systems.

A need arose for new platforms with properties of:

- Distribution
- Soft real-time properties
- Close to zero downtime (even across software updates and hardware faults)

Ericsson – backed by monopoly conditions and the IT hub of Kista – developed a platform to meet these needs.

It consisted of:

- The Erlang programming language
- The BEAM (distributed) virtual machine
- The Open Telecom Platform (OTP)

Joe Armstrong: Make it work, then make it beautiful, then if you really, really have to, make it fast. 90 percent of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful!

Who Uses BEAM?

Just another small runtime that no-one will ever use?

Well, BEAM (e.g., Erlang and Elixir) has a few users, including:

- **Discord** is using Elixir to scale to 5M concurrent users.
- **Facebook** uses Erlang to power its chat service serving 100M active users.
- **WhatsApp** uses Erlang to run messaging servers, each covering 2M users.
- **Ericsson** uses Erlang for its GPRS, 3G, 4G and 5G infrastructure, and has a market share of 40%.
- In 2018 **Cisco** shipped 2M devices running Erlang. At that point 90% of all internet traffic went through Erlang based nodes.
- 100+ NEPs (network equipment providers) – including the top 8 – use Erlang based components in their products. Same numbers for SPs (service providers).
- **A.P. Møller - Maersk** Measures, calculates, tracks and documents shipping-related CO_2 emissions.

Why BEAM?

Basically, if someone is both successful and doing any kind of routing that requires scalability, availability and consistently low latency, they are likely relying on BEAM.

But why?

Its simply the best stack for the job, in particular:

- It's battle tested
- It has mature tooling
- It has unrivaled performance (in the KPIs that matter in this space):
    - Robust low latency (high responsiveness)
    - Fair balancing of large numbers of concurrent tasks
    - Long-term real-live availability
        - Hot code updates
        - Robust to faults in hard- and software

(Robert) Virding's First Rule of Programming: Any sufficiently complicated concurrent program in another language contains an ad hoc informally-specified bug-ridden slow implementation of half of Erlang.

Runtime

"The thing is, your runtime is your foundation. It is your ground, and when your ground is solid then you can move on top of it with a lot of confidence at a steady reasonable pace, and you can start building your system and comfortably watch them grow in a way beyond your original plan and imagination. And this is why I would say that, when choosing your tool for the job, especially if the job is a job of building a software system, then ask not only what your language syntax, your library and frameworks can do for you. But first and foremost ask what your runtime can do for you, because it can make a huge difference."

— Saša Jurić

Why Elixir?

Elixir is a modern language that is executed on the BEAM. Unlike JVM and CLR, BEAM is a *distributed* virtual machine.

Typical reasons for picking Elixir for a project:

- It is designed specifically for the *actor model*.
  Design principle: *Concurrency as a 1st class citizen*
- It has a well documented standard library.
  Design principle: *Documentation as a 1st class citizen*
- It provides very good opportunities for testing.
  Design principle: *Testing as a 1st class citizen*
- The developer experience is considered key.
  Design goal: *Developer happiness*
- It can be programmed through *notebooks*.

Why Elixir? ▷ Use

Okay, so elixir has a few strengths, but is it just another esoteric language or are people actually using it for real work?

Why Elixir? ▷ Industry

- ***Network*** Elixir has, through BEAM, inherited a unique foundation for building distributed systems. Existing Erlang code can, using Elixir, be reused in a modern language. Because of this, many Erlang teams are starting to introduce Elixir code or completely shift to Elixir.

  Places where you appreciate:
  - Robustness
  - Little-to-no downtime
  - Easy-mode concurrency
  - Scalability
- ***Web*** Elixir is becoming popular for creating collaborative single-page webapps using the modern frameworks Phoenix and LiveView.
- ***Embedded*** The Nerves framework is a mature framework for writing, deploying and maintaining embedded code for Raspberry Pi class devices at scale.
- ***Finance*** The finance sector has begun adopting Elixir.

Why Elixir? ▷ Domains

Also:

- ***Adobe*** Combined client/cloud application for collaborative photography workflow.
- ***Live eSport statistics*** Copenhagen-based company.
- ***Slack***
- ***Motorola*** in Copenhagen
- ***InfluxData***
- ***Bleacher Report***
- ***Undead Labs***
- ***Pepsico***
- ***Heroku***
- ***FarmBot***

`https://elixir-companies.com` `https://elixir-lang.org/cases.html`

Part 2:
Functional Programming

Imperative vs Declarative Programming

Imperative Programming

In imperative programming, you focus on data. Code is expressed as an recipe where the ingredients are data. You tell the computer which specific steps to that and how they should be executed.

When programming imperatively, you describe *how* the result is reached.

Declarative Programming

I declarative programming, you focus on the result. The control flow is *not* described, at least not explicitly. Instead you define what the result is.

When programming declaratively, you describe *what* the program shall achieve.

Concept

Functional programming – or function-oriented programming – is a form of declarative programming.

Properties:

- Data are immutable
  - One can *rebind* variables.
  - Purely functional data structures have *persistence*, the property of keeping previous versions of the data structure unmodified.
- Everything is an expression (and is thus evaluated to a value)
  - An `if-else` construction evaluates to the value of either the true or the false block.
- Functions as values
  - **Example:** `incr` has the type `int` $\mapsto$ `int`.
- Functions are *composable* through their arguments.
- Function calls are cheap (at least when expressed to allow tail recursion).
- In principle, there are no loops
  - Instead, you recurse.
  - Functions are trees of expressions.

Concept ▷ Functions as Structural Elements

"Functions as first class citizens"

A function is a value. Accordingly:

- Names (variables) can be bound to it.
- It can be passed as an argument to functions (it self and others).
- Can be returned from functions.

Composability: Functionality is implemented through functions which behavior depend on other functions that they take as parameters.

Pure Functional Languages

A functional language is said to be *pure* if it only supports pure functions. Those are functions whose results depends only on the values of their parameters. This meas that they cannot have or rely on side-effects (neither memory or IO) or random number generators.

Pure functions are *idempotent*.

If two functions don't have side-effects, then they cannot affect each other and can thus trivially be executed in parallel. It is also very easy to test and compile pure functions.

The property, however, often makes it hard to write code. At least if you are not used to it.

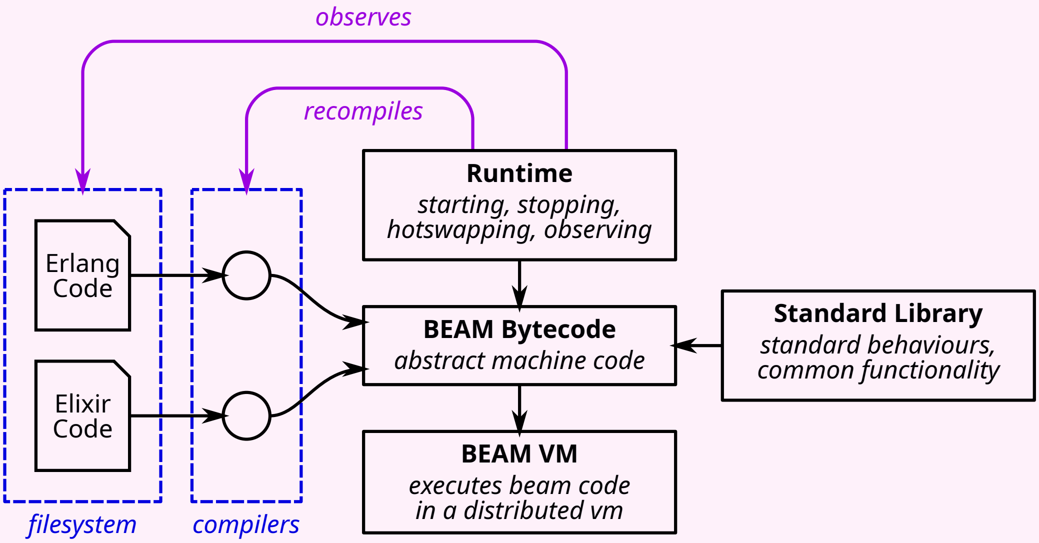
Because of this, language designers often avoid this property, or – at least – weaken it.

Functional programmers: *Exist*

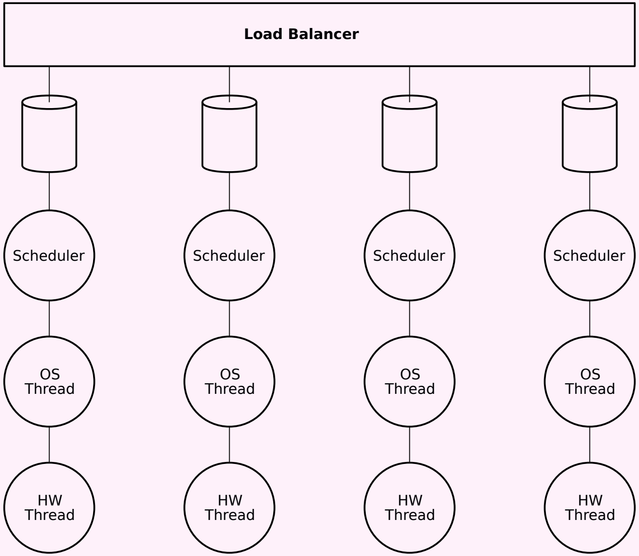
OOP programmers:

Part 3:
Language Walkthrough

BEAM


observes
recompiles
Runtime
starting, stopping, hotswapping, observing
Erlang Code
Elixir Code
BEAM Bytecode
abstract machine code
Standard Library
standard behaviours, common functionality
BEAM VM
executes beam code in a distributed vm
filesystem
compilers

BEAM ▷ Scheduler



Hello, World!

> *"When the first caveman programmer chiseled the first program on the walls of the first cave computer, it was a program to paint the string 'Hello, world' in Antelope pictures. Roman programming textbooks began with the 'Salut, Mundi' program. I don't know what happens to people who break with this tradition, but I think it's safer not to find out."*
> – The Linux Kernel Module Programming Guide (2007-05-18 ver 2.6.4)

So, lets have a go at it.

However, we have three options:

1. A module that compiles to bytecode.
2. Interactive mode (a REPL) that is interpreted.
3. Notebooks with LiveBook.

Hello, World! ▷ Module

Elixir code can be organized in modules.

```elixir
defmodule Hello do
  def hello do
    IO.puts "Hello, World!"
  end
end
```

Elixir comes with the `mix` build and project management tool.

This is how real projects are built, deployed and managed.

Hello, World! ▷ Interactive

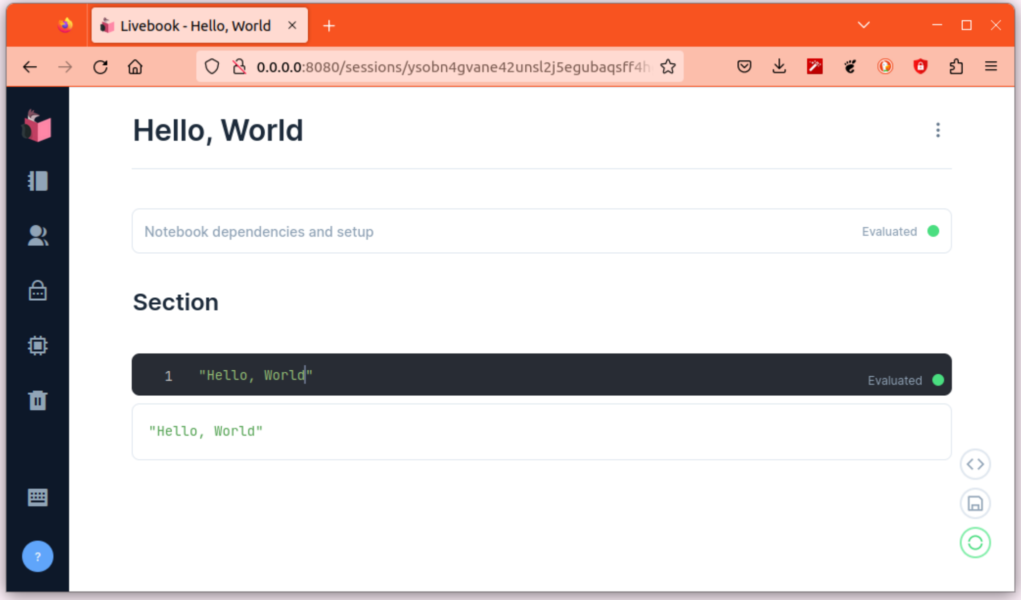
Like Python, Elixir has an interactive mode. The interpreter command is called `iex`.

```
iex(1)> IO.puts("Hello, World!")
Hello, World!
:ok
iex(2)> import IO
IO
iex(3)> puts("Hello, World!")
Hello, World!
:ok
```

When started using `iex -S mix` it gains access to compiled code.

One can also use `iex` to hook into a running cluster and interact with and/or modify the code running there.

Hello, World! ▷ Notebooks with LiveBook



Hello, World

Notebook dependencies and setup Evaluated

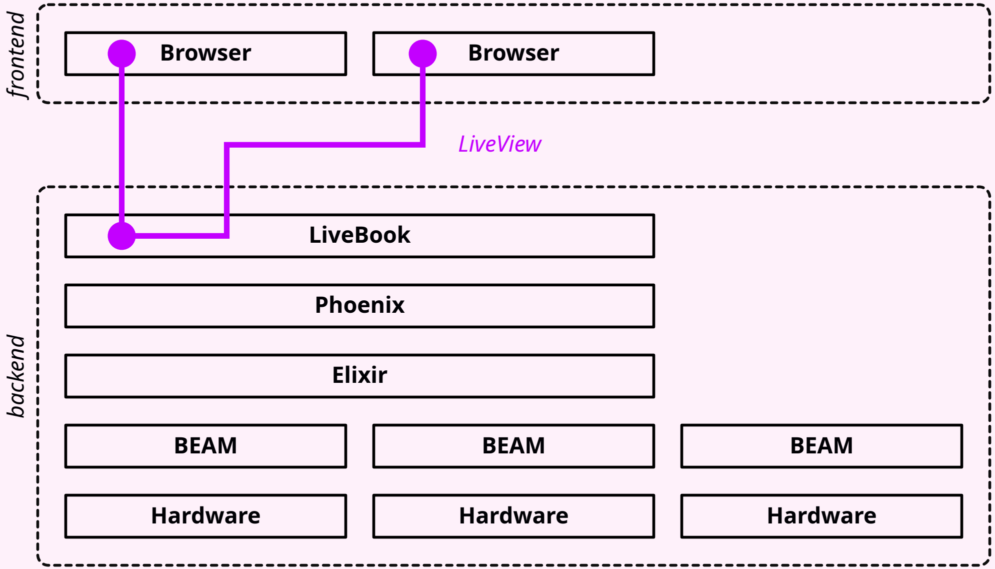
Section

```
"Hello, World"
```

Evaluated

```
"Hello, World"
```

Hello, World! ▷ Notebooks with LiveBook



Syntax

Comments with `#`

No statement separation character (aka no `;` at end of line).

Everything is an expression, and thus has a value.

Blocks:

- Begins with a keyword.
- Ends with `end`.

Visual ques that benefit from a font with ligatures:

- `|>` becomes ▷
- `->` becomes →

Basic Types

- ***integer*** Arbitrary size. Division results in a `float`. Use `div` and `rem` for integer division and remainder. Supports binary, octal and hexadecimal representations.
- ***float*** 64-bit IEEE floats.
- ***boolean*** Values are called `true` and `false`.
- ***atom*** Essentially a global `enum`. Atoms are names that start with a colon. They are equal if their names are equal. *Not garbage collected!*
- ***string*** Uses UTF-8 and double quotes. There is also a `charlist` that uses single quotes, but you rarely use it directly.
- ***list*** Immutable linked list. The head and tail of a list is accessible throught `hd` and `tl` functions.
- ***tuple*** Essentially a immutable array, declared with curly brackets.
- ***map*** An any term to any term mapping.
- ***pid*** A process ID.

Modules

```elixir
defmodule Example do
  @greet "Hello"

  defp greeting(name) do
    "#{@greet}, #{name}!"
  end

  def print_greeting(name) do
    IO.puts(greeting(name))
  end
end
```

Pattern Matching ▷ Basics

Assignments (bindings, really) in Elixir are done using the match operator:

$$\mathit{\$PATTERN} = \mathit{\$EXPRESSION}$$

```
iex(1)> i = 42
42
iex(2)> 42 = i
42
iex(3)> 56 = i
** (MatchError) no match of right hand side value: 42
iex(4)> _ = i
42
```

Underscores (_) and names starting with underscores act as don't-cares.

Pattern Matching ▷ Tuples, Lists and Maps

```
iex(1)> {n, _, e, _} = Circle.get_bbox(circle)
{1, 2, 3, 4}
iex(2)> [r, g, b] = [0.1, 0.8, 0.8]
[0.1, 0.8, 0.8]
iex(3)> r
0.1
iex(4)> [head|tail] = ["Bonobo", "Gorilla", "Gibbon"]
["Bonobo", "Gorilla", "Gibbon"]
iex(5)> head
"Bonobo"
iex(6)> tail
["Gorilla", "Gibbon"]
iex(7)> coordinate = %{x: 3.14, y: 2.7, z: -1}
%{x: 3.14, y: 2.7, z: -1}
iex(8)> %{x: x_value, y: y_value} = coordinate
%{x: 3.14, y: 2.7, z: -1}
iex(9)> x_value
3.14
```

Pattern Matching ▷ Conditionals

Pattern matching fit into conditionals:

```
iex(1)> case Internet.download("https://www.sdu.dk") do
...(2)>   {:ok   , message} -> "Success: #{message}"
...(3)>   {:error, message} -> "Error: #{message}"
...(4)> end
"Success: Downloaded https://www.sdu.dk"
```

The standard library and frameworks are designed for you to take advantage of this.

Pattern Matching ▷ Nesting

Patterns can be nested:

```
defmodule Worker do
  def work([]) do
    :ok
  end

  def work([{a,b}|tail]) do
    IO.puts(a-b)
    work(tail)
  end
end
```

Example:

```
iex(1)> Worker.work([{1,2},{1,3},{1,4}])
-1
-2
-3
:ok
```

PATTERN MATCHING
LIGHTYEAR
SPACE RANGER
PATTERN MATCHING EVERYWHERE

Pipelines

```
a = input("your name?")
b = lookup(a, query)
c = process(b)
d = report(c)
```

Can be written as:

```
d =
  input("your name?")
   ▷ lookup(query)
   ▷ process()
   ▷ report()
```

Pipelines ▷ Postscript

The pipe operator is written as `|>`, but given the use of a font that uses ligatures it can be rendered as ▷ .

In order to make frameworks function neatly it is convention to add a bang-variant to functions that return a okay/error tuple. Such a function will either succeed and return an unwrapped value, or error out (functionality that is similar to exceptions).

Example:

- `readline` returns a tuple wherein the first element is either `:ok` or `:error`, and the second element contains either the line read, or – in case of an error – the error message.
- `readline!` Either produces the line read, or errors out.

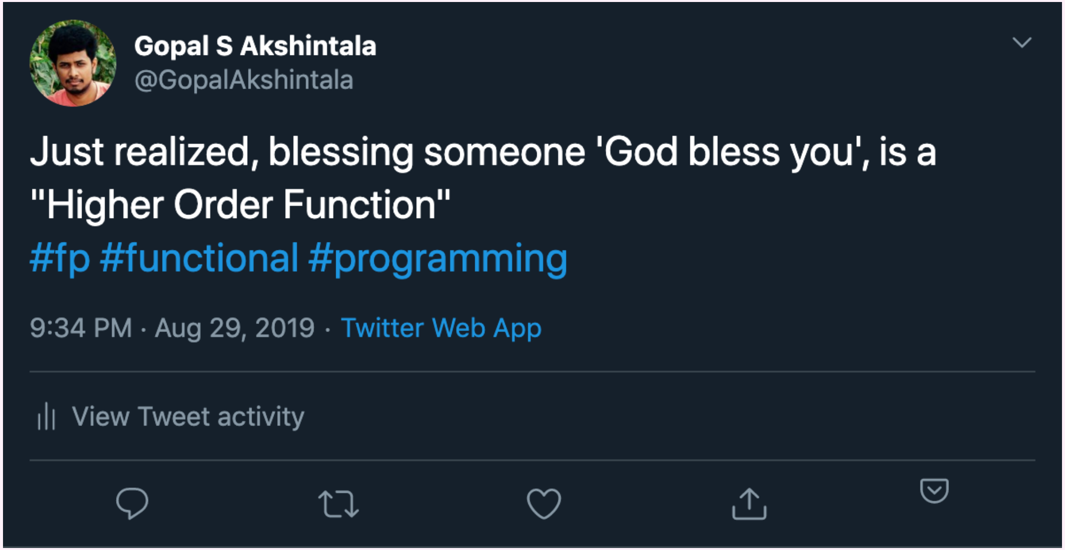
Anonymous Functions

```
iex(1)> sum = fn (a,b) -> a + b end
#Function<43.65746770/2 in :erl_eval.expr/5>
iex(2)> sum.(1, 2)
3
iex(4)> same = fn
...(4)>   (n, n) -> true
...(4)>   (_, _) -> false
...(4)> end
iex(5)> same.(1,2)
false
iex(6)> same.(1,1)
true
```

Higher-Order Functions

Functions can take other functions as parameters:

```
iex(1)> even? = fn value -> rem(value, 2)==0 end
#Function<42.125776118/1 in :erl_eval.expr/6>
iex(2)> incr = fn value -> value+1 end
#Function<42.125776118/1 in :erl_eval.expr/6>
iex(3)> list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
iex(4)> list |> Enum.filter(even?)
[0, 2, 4, 6, 8, 10]
iex(5)> list |> Enum.filter(even?) |> Enum.map(incr)
[1, 3, 5, 7, 9, 11]
```



Gopal S Akshintala
@GopalAkshintala

Just realized, blessing someone 'God bless you', is a "Higher Order Function"
#fp #functional #programming

9:34 PM · Aug 29, 2019 · Twitter Web App

Messages

```
send(self(), "Hello, you are looking good today!")

receive do
  message -> IO.puts(message)
end
```

Processes

```
pid =
  spawn(fn ->
    receive do
      message -> IO.puts(message)
    end
  end)

send(pid, "Hello, you are looking good today!")
```

Part 4:
The Actor Model

Definition

An *actor* (or *process* in Elixir lingo) consists of:

- State
- Functionality
- A message queue

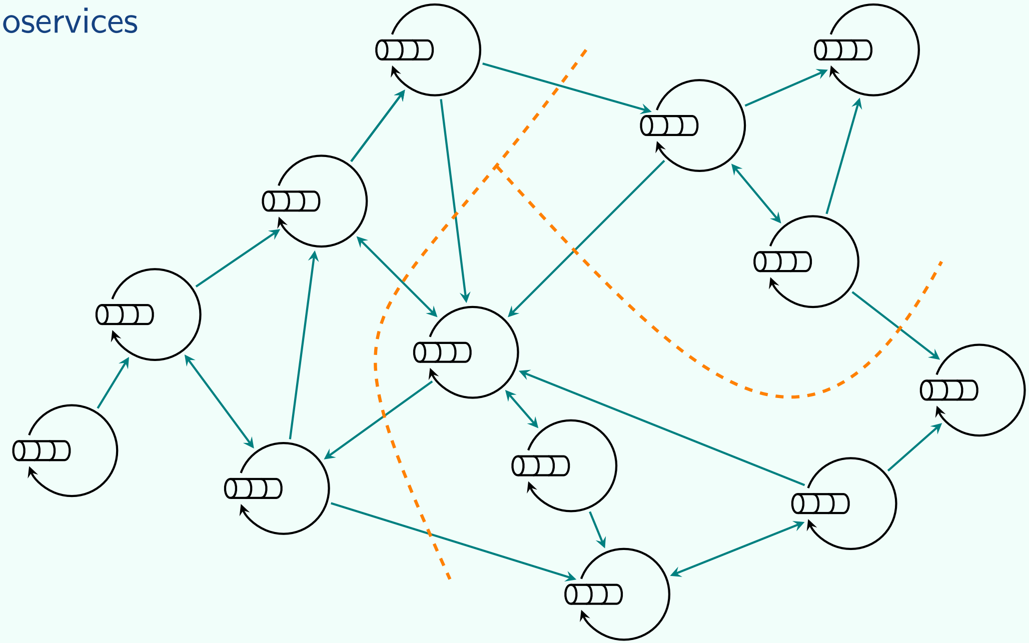
Actors are isolated from each other, run concurrently and communicate through message passing.

Elixir supports OTP actors for which it inherits a lot of tooling.

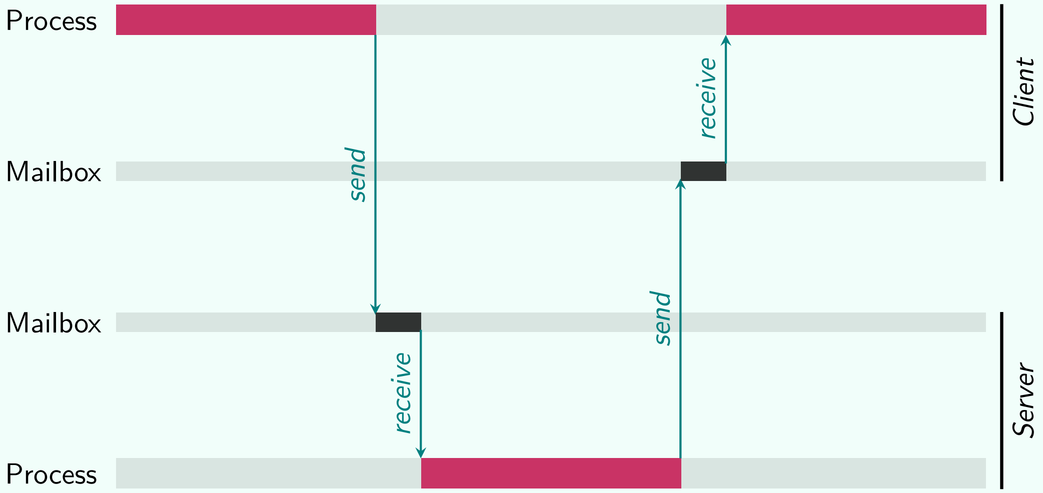
Concurrency model:

- Actors are inactive as long as their inbox is empty.
- Messages are consumed FIFO-style and processed strictly sequentially†.
- The *unit of concurrency* is an actor.

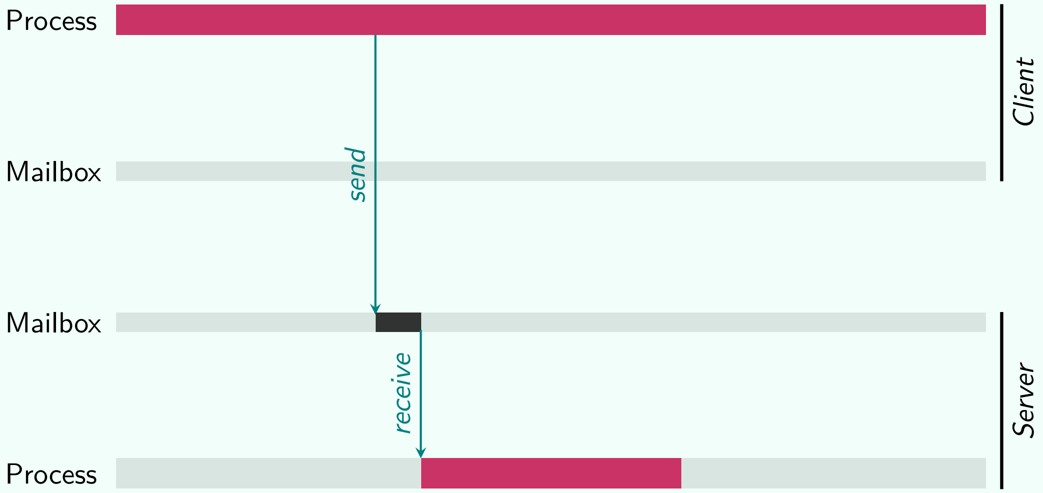
Nanoservices



Communication ▷ Synchronous Communication (call)



Communication ▷ Asynchronous Communication (cast)



Process Example

Process Example ▷ Definition

```elixir
defmodule State do
  use GenServer

  # (client) interface

  def start_link() do
    GenServer.start_link(__MODULE__, nil)
  end

  def get(server) do
    GenServer.call(server, {:get})
  end
  def set(server, value) do
    GenServer.cast(server, {:set, value})
  end

  # callbacks

  def init(_) do
    {:ok, 0} # 0 is the initial state
  end

  def handle_call({:get}, _from, state) do
    {:reply, state, state}
  end
  def handle_cast({:set, value}, _state) do
    {:noreply, value}
  end

end
```

Process Example ▷ Use

```
iex(1)> {:ok, server} = State.start_link()
{:ok, #PID<0.181.0>}
iex(2)> State.get(server)
0
iex(3)> State.set(server, 42)
:ok
iex(4)> State.get(server)
42
```

Process Example ▷ Complex Parameters

Lets extend the definition of our `State` server with:

```
...

  # (client) interface

  def manipulate(server, function) do
    GenServer.cast(server, {:manipulate, function})
  end

  # callbacks

  def handle_cast({:manipulate, function}, state) do
    {:noreply, function.(state)}
  end

...
```

Process Example ▷ Complex Parameter Use

```
iex(1)> {:ok, server} = State.start_link()
{:ok, #PID<0.191.0>}
iex(2)> State.set(server, 2.7)
:ok
iex(3)> State.get(server)
2.7
iex(4)> area = fn r -> 3.14*r*r end
#Function<44.65746770/1 in :erl_eval.expr/5>
iex(5)> State.manipulate(server, area)
:ok
iex(6)> State.get(server)
22.890600000000006
iex(7)> oopsie = fn r -> r/0 end
#Function<44.65746770/1 in :erl_eval.expr/5>
iex(8)> State.manipulate(server, oopsie)
:ok
iex(9)>
14:14:51.447 [error] GenServer #PID<0.191.0> terminating
** (ArithmeticError) bad argument in arithmetic expression
    :erlang./(22.890600000000006, 0)
    iex:38: State.handle_cast/2
    (stdlib 3.17.1) gen_server.erl:695: :gen_server.try_dispatch/4
    (stdlib 3.17.1) gen_server.erl:771: :gen_server.handle_msg/6
    (stdlib 3.17.1) proc_lib.erl:226: :proc_lib.init_p_do_apply/3
Last message: {:"$gen_cast", {:manipulate, #Function<44.65746770/1 in :erl_eval.expr/5>}}
State: 22.890600000000006
```

Supervision Trees ▷ Sandboxed Processes

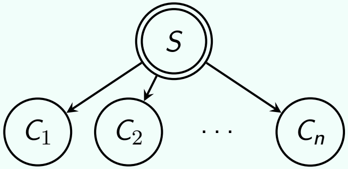
Each process operates in a *sandbox*.

If something goes wrong, it will *crash* within the sandbox.

The crash of a process will thus not (directly) affect any other process.

However, a crash of a process can be detected by another process, and this process can act on such a crash.

Supervision Trees ▷ Supervision


$$\begin{tikzcd} & S \arrow[dl] \arrow[d] \arrow[drr] & & \\ C_1 & C_2 & \cdots & C_n \end{tikzcd}$$

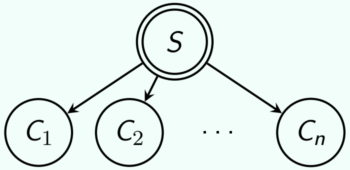
A *supervisor* is a type of process that is responsible for administrating the lifecycles of a number of processes.

It knows how to start them, and how to stop them (correctly).

It is the job of a supervisor to start and stop these processes, and to handle any crashes among the supervised processes by restarting the affected processes.

But which ones are affected?

Supervision Trees ▷ Restart Strategies



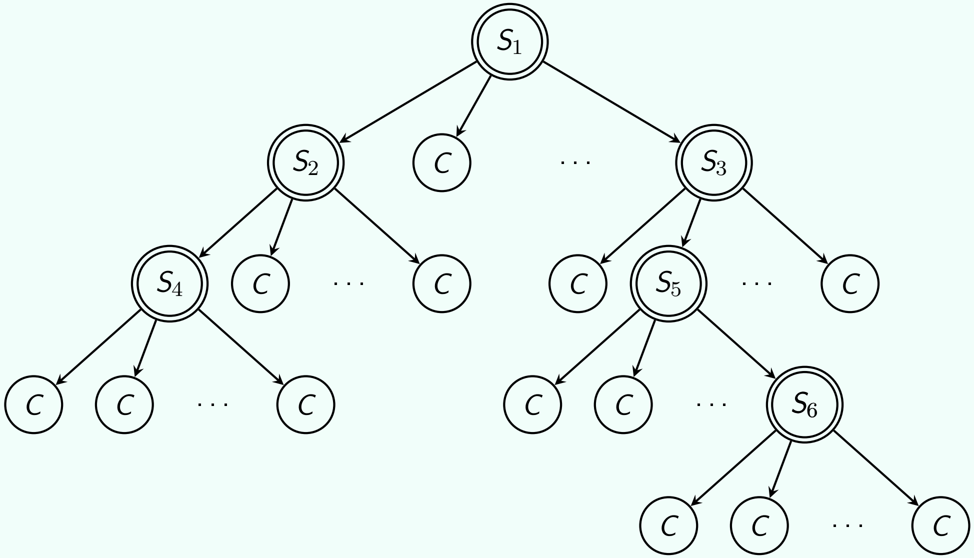
$$\begin{tikzcd}
 & & S \arrow[dll] \arrow[dl] \arrow[dr] & \\
C_1 & C_2 & \cdots & C_n
\end{tikzcd}$$

Strategies:

- ***One for One*** If one *child process* fails then only this process is restarted.
- ***One for All*** If one child process fails then all child processes are restarted.
- ***Rest for One*** If one child process fails then it and all child processes that are listed later in the order of child processes are restarted. If process C_3 fails then processes $C_3 \ldots C_n$ are restarted (where C_n is the last process).

Means are provided to define other and more complex strategies.

Supervision Trees ▷ Complex Behavior



Supervision Trees ▷ Example

```elixir
defmodule State do
  use GenServer

  # (client) interface

  def start_link(_) do
    GenServer.start_link(__MODULE__, nil,
                         name: :mystate)
  end

  def get() do
    GenServer.call(:mystate, {:get})
  end
  def set(value) do
    GenServer.cast(:mystate, {:set, value})
  end
  def manipulate(function) do
    GenServer.cast(:mystate, {:manipulate, function})
  end

  # callbacks

  def init(_) do
    {:ok, 0} # 0 is the initial state
  end

  def handle_call({:get}, _from, state) do
    {:reply, state, state}
  end
  def handle_cast({:set, value}, _state) do
    {:noreply, value}
  end
  def handle_cast({:manipulate, function}, state) do
    {:noreply, function.(state)}
  end

end
```

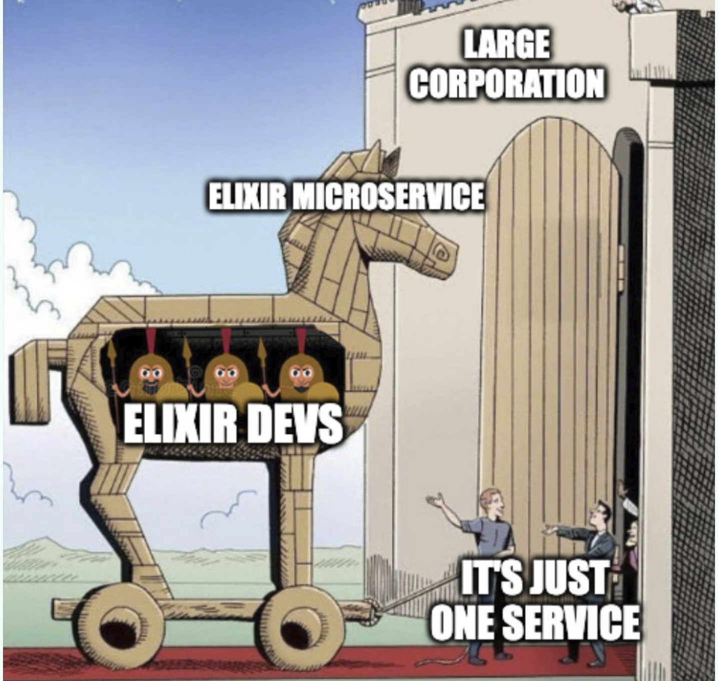
Modified to bind the name `:mystate` to the PID of the server.

Supervision Trees ▷ Example

```
defmodule StateSupervisor do
  use Supervisor

  def start_link(opts\\[]) do
    Supervisor.start_link(__MODULE__, :ok, opts)
  end

  def init(:ok) do
    children = [
      State
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

Supervision ▷ Example

```
iex(1)> StateSupervisor.start_link()
{:ok, #PID<0.165.0>}
iex(2)> State.set(2.7)
:ok
iex(3)> State.get()
2.7
iex(4)> area = fn r -> 3.14*r*r end
#Function<44.65746770/1 in :erl_eval.expr/5>
iex(5)> State.manipulate(area)
:ok
iex(6)> State.get()
22.890600000000006
iex(7)> oopsie = fn r -> r/0 end
#Function<44.65746770/1 in :erl_eval.expr/5>
iex(8)> State.manipulate(oopsie)
:ok
iex(9)>
22:13:49.677 [error] GenServer :mystate terminating
** (ArithmeticError) bad argument in arithmetic expression
    :erlang./(22.890600000000006, 0)
    iex:33: State.handle_cast/2
    (stdlib 3.17.1) gen_server.erl:695: :gen_server.try_dispatch/4
    (stdlib 3.17.1) gen_server.erl:771: :gen_server.handle_msg/6
    (stdlib 3.17.1) proc_lib.erl:226: :proc_lib.init_p_do_apply/3
Last message: {:"$gen_cast", {:manipulate, #Function<44.65746770/1 in :erl_eval.expr/5>}}
State: 22.890600000000006

nil
iex(10)> State.set(2.7)
:ok
iex(11)> State.get()
2.7
```


LARGE CORPORATION
ELIXIR MICROSERVICE
ELIXIR DEVS
IT'S JUST ONE SERVICE

Part 5:
When Things Go Wrong

Let It Crash!

When things do go wrong, it is almost always an issue with state.

We don't know what is wrong; just that something is wrong. Our options are limited.

In Elixir code, that state will reside in a sandboxed process.

A well structured Elixir program or system will already have supervisors configured to deal with crashes.

So, often the best choice ends up simply letting the process crash so that its supervisor will reload it with a fresh state.

That will also save the millions of other processes which are doing beneficial work with their (correct) state. No need to interfere with them.

Deffensive vs Offensive Programming

Defensive Programming

Some actions may fail for more or less well-defined reasons. This can be handled by a combination of:

- Handling of error codes using if statements
- Exception handling

Common for these is that you end up sprinkling error handling across the core logic of ones code, and that makes it hard to read.

Offensive Programming

With a supervision tree, there is an alternative:
"Let it fail!"

It is okay that your code fails. A supervisor will catch and restart it.

YOU WORK AROUND INVALID INPUT
I FAIL FAST AND VISIBLY AT THE FIRST SIGN OF ONE
WE ARE NOT THE SAME
imgflip.com

Part 6:
Nerves

Framework

Nerves is a framwork for programming Raspberry Pi class (and up) hardware.

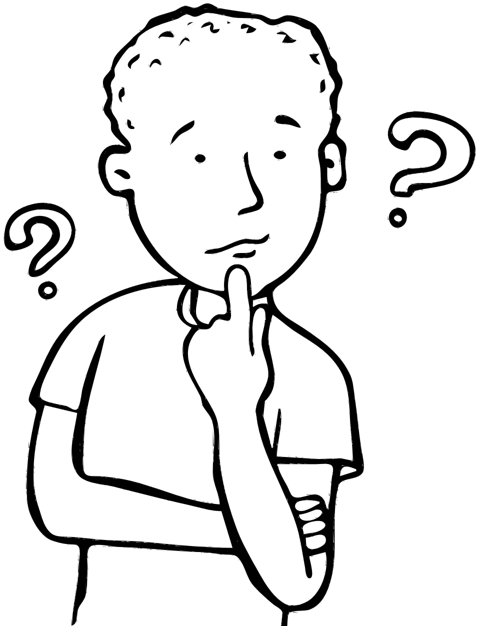
It builds on a minimal Linux kernel and boots straight into a BEAM virtual machine.

Infrastructure exists in the form of NervesHub for sending out firmware updates.

This means that

1. You have the option of having a cluster of (software) virtual machines that spans cloud, fog and edge.
2. That logic can be shifted between these layers as needed.
3. That the interfaces between these layers are indistinguishable from your native API (instead of having to rely on a heavy and/or inflexible transport).
4. That you can easily move logic to the data.

Questions?



https://openclipart.org/detail/238687/boy-thinking-of-question