Welcome to Elixir

“This is good shit“
— Joe Armstrong

June 23, 2016

Table of Contents

Overview

Power of Erlang

Elixir basics

Macros

Tooling and Abstractions

Motivation

- Power of Erlang
- Elixir basics
- Macros
- Tooling and abstractions

Table of Contents

Overview

Power of Erlang

Elixir basics

Macros

Tooling and Abstractions

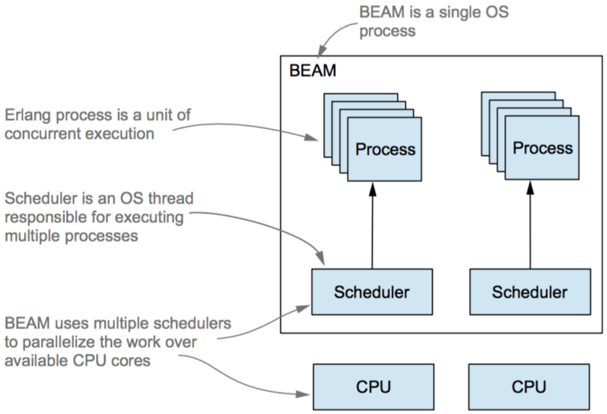
Erlang

- created in mid-1980s
- designed for telecom
- connect multiple systems
- minimal impact of errors
- entire system should never go down

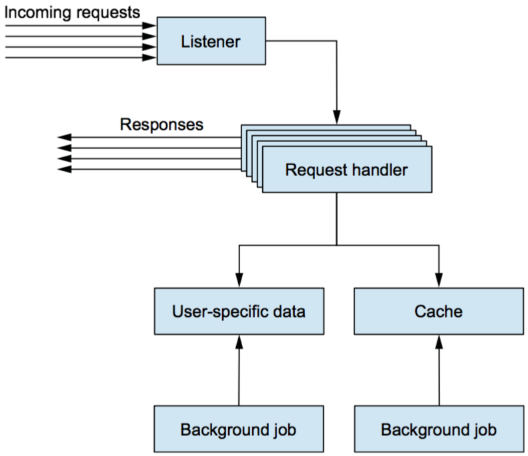
High availability

- fault tolerance
- scalability
- distribution
- responsiveness
- live update

How do they do it?


BEAM is a single OS process
BEAM
Erlang process is a unit of concurrent execution
Process
Process
Scheduler is an OS thread responsible for executing multiple processes
Scheduler
Scheduler
BEAM uses multiple schedulers to parallelize the work over available CPU cores
CPU
CPU

Server side systems



A modern system

Technical requirements	Server
HTTP server	Nginx and Phusion Passenger
Request processing	Ruby on Rails
Long-running requests	Java and Go
Server-wide state	Redis
Persistable data	Redis and MongoDB
Background jobs	Cron, Bash scripts, and Ruby
Service crash recovery	Upstart

OTP

Technical requirements	Server
HTTP server	Erlang
Request processing	Erlang
Long-running requests	Erlang
Server-wide state	Erlang
Persistable data	Erlang
Background jobs	Erlang
Service crash recovery	Erlang

Table of Contents

Overview

Power of Erlang

Elixir basics

Macros

Tooling and Abstractions

Syntax

“Elixir syntax is like a marriage of DSL friendly Ruby and the powerful hygenic macros of Clojure.“
– Devin Torres

The basics you know

```
1 + 1               # => 2
2 * (3 + 1) / 4     # => 2.0
1 + 2; 1 + 3        # => 4

greeting = "Hello␣World!"
IO.puts(greeting)
# => Hello World!
# => :ok
```

Modules and function, oh my!

```
defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end
end
```

Composing functions

```
def process_xml(model, xml) do
  model
  |> update(xml)
  |> process_changes
  |> persist
end
```

Function arity

```
defmodule Rectangle do
  def area(a), do: area(a, a)
  def area(a, b), do: a * b
end
```

Destructuring

```
def do_something({:ok, value}) do
  # use the value here
end

def do_something({:warning, value}) do
  # warn user before proceeding
end

def do_something({:error, message}) do
  # produce a nice error message to the user
end
```

Typespec

```
defmodule Circle do
  @pi 3.14159

  @spec area(number) :: number
  def area(r), do: r * r * @pi

  @spec circumference(number) :: number
  def circumference(r), do: 2 * r * @pi
end
```

Table of Contents

Overview

Power of Erlang

Elixir basics

Macros

Tooling and Abstractions

Macros

- code transformation at compile time
- code that can change semantics of the input
- a lot of elixir functionality is implemented in macros *e.g. def, unless*
- should be used sparingly

Macro expension

```
unless some_exp do
  block_1
else
  block_2
end
```

```
if some_exp do
  block_2
else
  block_1
end
```

Quoting

```
quote do: sum [1, 2, 3]
# => { :sum, [], [1, 2, 3] }

quote do: 1 + 1
# => { :+,
#       [context: Elixir, import: Kernal],
#       [1, 1] }
```

Simple macro

```
defmacro match?(left, right) do
  quote do
    case unquote(right) do
      unquote(left) ->
        true
      _ ->
        false
    end
  end
end
```

Usine our new macro!

```
list = [{:a,1},{:b,2},{:a,3}]
# => [a: 1, b: 2, a: 3]

Enum.filter list, fn (thing) do
  match?({:a, _}, thing)
end
# => [a: 1, a: 3]

Enum.filter list, match?({:a, _}, &1)
# => [a: 1, a: 3]
```

Table of Contents

Overview

Power of Erlang

Elixir basics

Macros

Tooling and Abstractions