

Programming Language

Elixir

Erick Henrique Dutra de Souza

Lucas Cota Dornelas

23 de agosto de 2021

Laboratório de Linguagens de Programação



Introdução

Elixir é uma linguagem de programação dinâmica, funcional e concorrente para a construção de aplicativos escaláveis e de fácil manutenção. Ela aproveita o Erlang VM, máquina virtual do Erlang, conhecido por executar sistemas de baixa latência, distribuídos e tolerantes a falhas. Elixir compila em cima de Erlang para fornecer aplicações distribuídas, em tempo real suave, tolerante a falhas, non-stop, mas também a estende para suportar metaprogramação com macros e polimorfismo via protocolos

O Elixir é usado com sucesso em desenvolvimento web, software embarcado, ingestão de dados e processamento de multimídia em uma ampla gama de indústrias.



Motivação

A Linguagem foi criada em um projeto de R&D da Plataformatec, uma subsidiária do Nubank, pelo brasileiro José Valim. Seus principais objetivos foram permitir uma maior extensibilidade e produtividade no Erlang VM, mantendo a compatibilidade com ferramentas e ecossistema de Erlang.

Sua principal característica que oferece uma ótima escalabilidade é que seu código é executado dentro de threads leves de execução (chamados de processos) que são isolados e trocam informações por meio de mensagens. Como principal ferramenta, Mix é uma ferramenta de construção que permite criar facilmente projetos, gerenciar tarefas, executar testes e muito mais.

Em 12 de julho de 2018, a HoneyPot lançou um mini-documentário sobre a linguagem Elixir.

Característica da linguagem

A linguagem Elixir possui diversas características que vale uma breve explicação para contextualizar os exemplos que usamos.

Átomo

No Elixir existe um tipo de dado que seu valor é seu nome. Ele é declarado começando com dois pontos ‘:’.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays the following Elixir code and its output:

```
iex(24)>  
nil  
iex(25)> :hello_world  
:hello_world  
iex(26)>
```

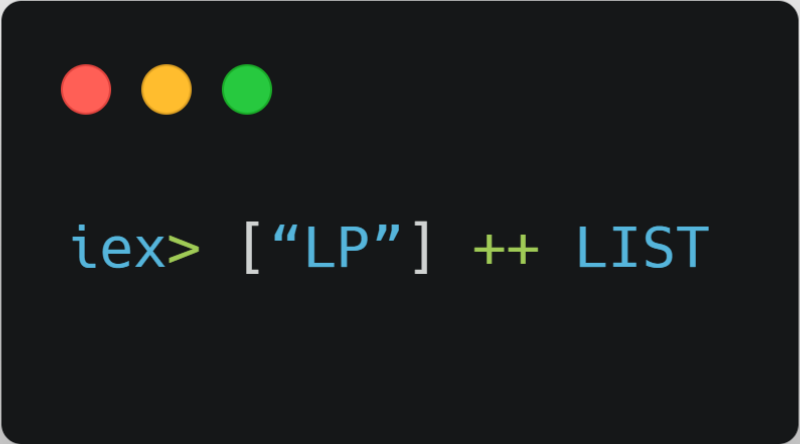
Coleções

As coleções que o Elixir implementa são Listas, Tuplas, Keyword List, Map.

- List

O Elix implementa o tipo como listas ligadas, oque faz com que acessar o tamanho da lista seja feito em um tempo linear ($O(n)$). Por isso é mais rápido inserir um elemento no começo da lista do que ao final dela.

a) Prepending (rápido):

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The text inside the terminal is `iex> ["LP"] ++ LIST`, where `iex>` is in light blue, `["LP"]` is in light blue, `++` is in yellow, and `LIST` is in light blue.

```
iex> ["LP"] ++ LIST
```


b) Appending (devagar):

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The text `iex> LIST ++ ["LP"]` is displayed in a light blue monospace font.

```
iex> LIST ++ ["LP"]
```

- Tuplas

Tuplas é um tipo de coleção de valores ordenados. As tuplas são armazenadas de maneira contínua em memória, o que faz com que acessar seu tamanho seja rápido, porém modificá-la é mais custosa. Uma tupla no Elixir normalmente tem 2 ou 3 elementos. Se for necessário mais do que isso, provavelmente uma lista será melhor.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The text `iex> {:ok, :live_started, :youtube}` is displayed in a light blue monospace font.

```
iex> {:ok, :live_started, :youtube}
```

- Keyword List

Listas de palavras chaves são coleções associativas do Elixir. As Keyword list nada mais são do que listas de tuplas.



```
iex> [foo: "bar", hello: "world"]  
# [foo: "bar", hello: "world"]  
  
iex> [{:foo, "bar"}, {:hello, "world"}]  
# [foo: "bar", hello: "world"]
```

- Map

Mapas normalmente são a escolha para armazenamento chave-valor. A diferença entre os mapas e as listas de palavras-chave está no fato de que os mapas permitem chaves de qualquer tipo e não seguem uma ordem.



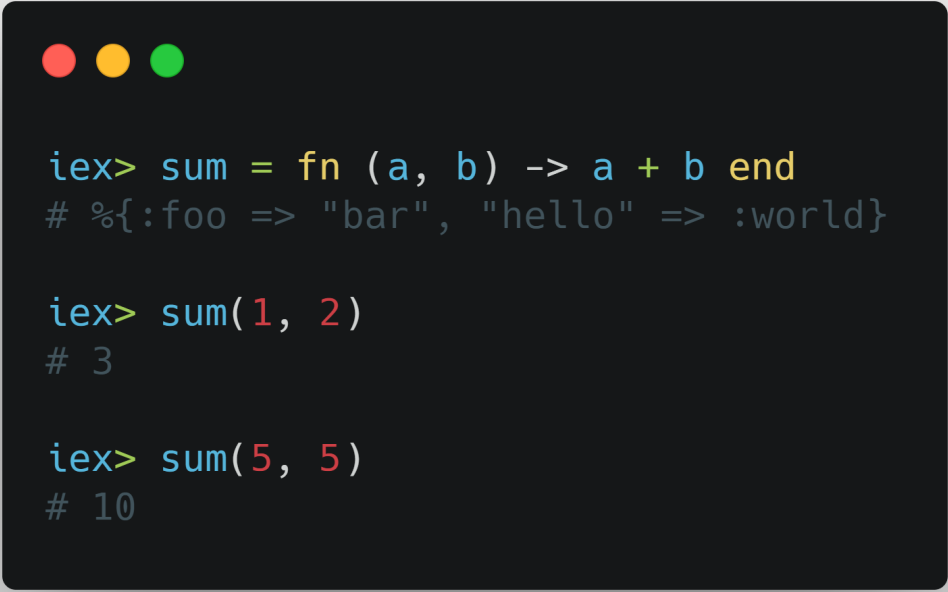
```
iex> map = %{:foo => "bar", "hello" => :world}  
# %{:foo => "bar", "hello" => :world}  
  
iex> map[:foo]  
# "bar"  
  
iex> map["hello"]  
# :world
```

Funções

As funções são cidadãos de primeira classe, isso significa que elas são tratados como qualquer outro tipo de dado.

- Funções anônimas

Uma função que não possui nome (ou identificador), ela normalmente é usada como argumento para outras funções.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays R code for defining and using an anonymous function.

```
iex> sum = fn (a, b) -> a + b end
# %{:foo => "bar", "hello" => :world}

iex> sum(1, 2)
# 3

iex> sum(5, 5)
# 10
```

- Funções nomeadas

As funções também podem ser recebidas e definidas dentro de módulos. Podemos especificar seu nome e seu número de argumentos.



```
defmodule Greeter do  
  def hello(name) do  
    "Hello, " <> name  
  end  
end
```

```
end
```

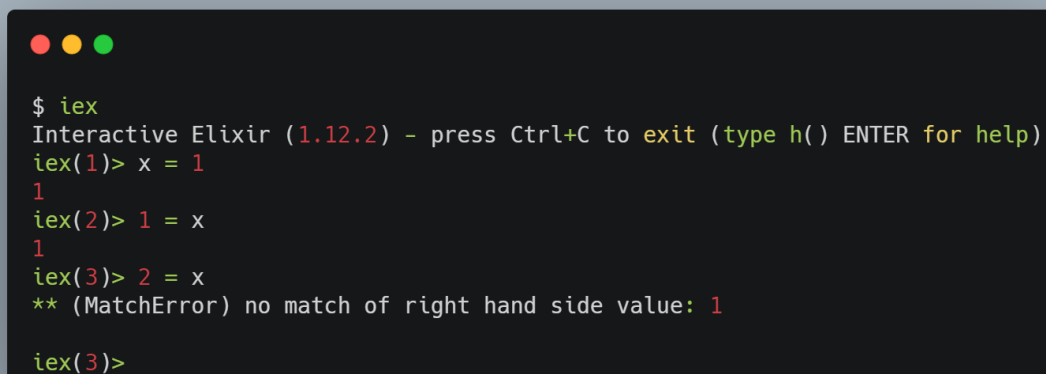
```
iex> Greeter.hello("Lucas")  
# Hello, Lucas
```

Pattern Matching

Essa funcionalidade é uma poderosa parte de Elixir que nos permite procurar padrões simples em valores, estruturas de dados, e até funções. Existem dois operadores que usam essa funcionalidade, o operador `' = '` é chamado match e o operador pin `' ^ '`.

- Match

Esse operador pode confundir um pouco pelo fato de normalmente usamos a mesma forma para atribuir um valor a uma variável, mas ele é 'entendível' como o sinal de igualdade da matemática. O Elixir tenta relacionar, por meio de padrões, as duas partes, caso a comparação não for possível ele retorna um erro.



```
$ iex
Interactive Elixir (1.12.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> x = 1
1
iex(2)> 1 = x
1
iex(3)> 2 = x
** (MatchError) no match of right hand side value: 1
iex(3)>
```

Contudo é possível re-associar uma relação já feita e ao comparar irá retornar corretamente a última relação.

```
iex(3)> x = 2
2
iex(4)> 2 = x
2
iex(5)> 1 = x
** (MatchError) no match of right hand side value: 2
```

- Pin

Muitas vezes não queremos o comportamento acima, para isso o operador pin foi criado. Quando fixamos a variável em associação ao valor existente ao invés de re-associar a um novo valor ele irá retornar um erro.

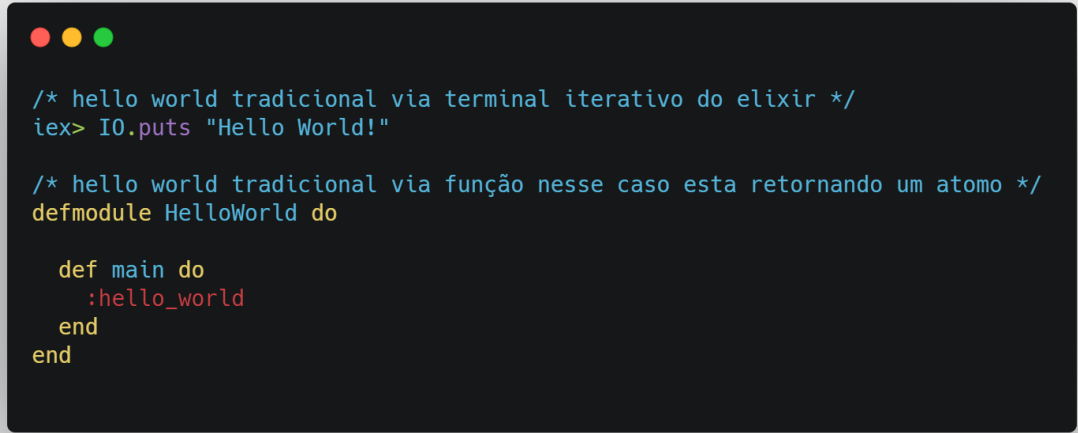
```
iex(18)> y = 1
1
iex(19)> ^y = 2
** (MatchError) no match of right hand side value: 2
```

Exemplos

Para criar os exemplos criamos um projeto que contém vários casos de exemplo, casos que aplicam as características da linguagem citadas na seção anterior.

1. Hello World:

Duas maneiras de retornar “Hello World” no elixir.



```
/* hello world tradicional via terminal iterativo do elixir */
iex> IO.puts "Hello World!"

/* hello world tradicional via função nesse caso esta retornando um atomo */
defmodule HelloWorld do

  def main do
    :hello_world
  end
end
```

2. Definição de um módulo com estrutura:

As estruturas do Elixir quando são criadas são retornadas a um mapa especial com um conjunto definido de chaves e valores padrões. Ele deve ser definido dentro de um módulo, no qual leva o nome dele. É comum para um struct ser a única coisa definida dentro de um módulo.

```
/* Definição de um modulo e de estrutura */  
defmodule Item do  
  defstruct name: nil, done: false  
end
```

3. Criação de um mapa com estrutura:

Definição de uma função nomeada e criação de uma estrutura.

```
/* Criação de uma função nomeada e criar uma mapa via estrutura */  
defmodule Item do  
  defstruct name: nil, done: false  
  
  /* Definição um valor padrão para a propriedade done como false */  
  def new(name, done \\ false) do  
    %Item{name: name, done: done}  
  end  
  
end
```

4. Operador para mapas:

Operador especial para manipular propiedades de mapas.

```
/* Operador para actualizar propiedades de mapas ya existentes*/
defmodule Item do
  defstruct name: nil, done: false

  def new(name, done \\ false) do
    %Item{name: name, done: done}
  end

  def done (item) do
    %{item | done: true}
  end

  def undone(item) do
    %{item | done: false}
  end
end
```

5. Desestruturando parâmetro:

Devido a imutabilidade não é possível atualizar uma variável, assim devemos pegar propriedades específicas para criar uma nova atualizando as propriedades.

```
/* Desestruturando mapa em uma propriedade */
defmodule Board do
  defstruct name: nil, items: []

  def new(name, items \\ []) do
    %Board{name: name, items: items}
  end

  def add_item_to_board(%{name: name, items: items}, new_item) do
    new(name, [new_item | items])
  end
end
```

6. Operador Pipe:

O operador pipe `|>` passa o resultado de uma expressão como o primeiro parâmetro de outra expressão. O pipe pega o resultado da esquerda e o passa para o lado direito.

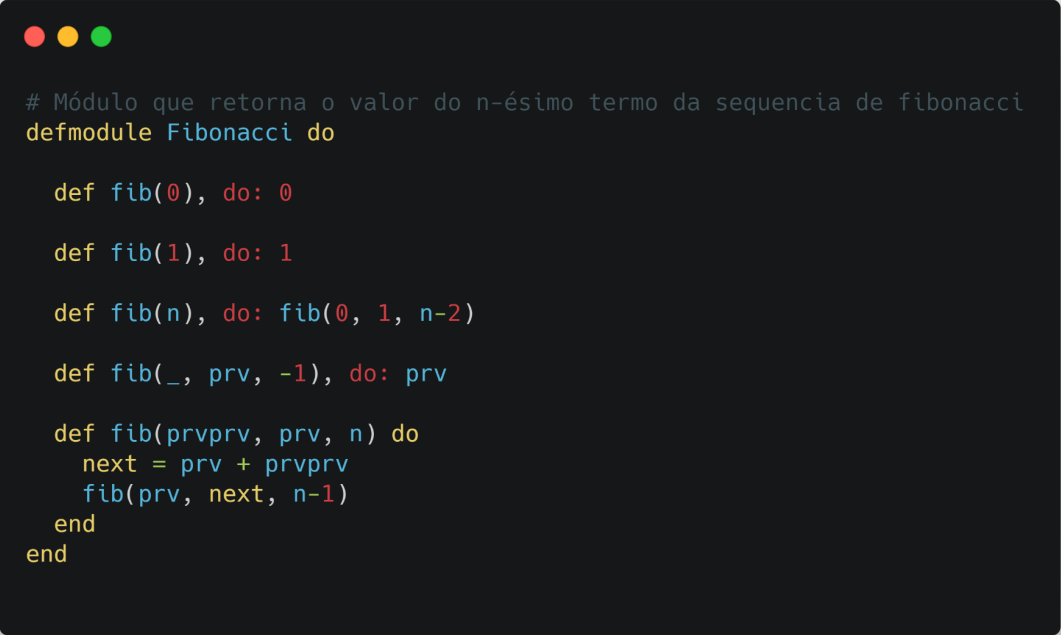
```
defmodule Board do
  defstruct name: nil, items: []

  # Operador Pipe
  def mark_item_as_done(board, done_item) do
    board
    |> remove_item_from_board(done_item)
    |> add_item_to_board(Item.done(done_item))
  end

  def mark_item_as_undone(board, undone_item) do
    board
    |> remove_item_from_board(undone_item)
    |> add_item_to_board(Item.undone(undone_item))
  end
end
```


7. Funções nomeadas e aridade de funções:

As funções são nomeadas pela combinação do nome e aridade (quantidade dos argumentos) das funções.



```
# Módulo que retorna o valor do n-ésimo termo da sequencia de fibonacci
defmodule Fibonacci do

  def fib(0), do: 0

  def fib(1), do: 1

  def fib(n), do: fib(0, 1, n-2)

  def fib(_, prv, -1), do: prv

  def fib(prvprv, prv, n) do
    next = prv + prvprv
    fib(prv, next, n-1)
  end
end
```

8. Funções privadas:

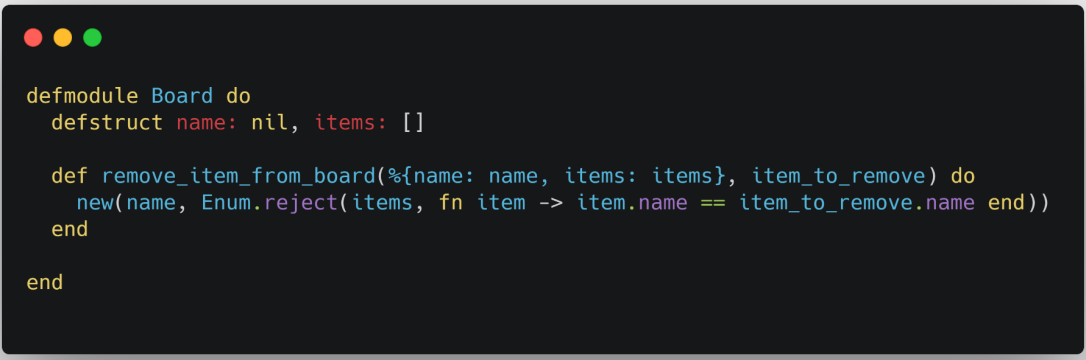
Quando não quisermos que outros módulos acessem uma função específica, nós podemos torná-la uma função privada, que só podem ser chamadas dentro de seus módulos. Nós podemos defini-las em Elixir com ***defp***.

```
defmodule Anagram do
  def anagrams?(a, b) when is_binary(a) and is_binary(b) do
    sort_string(a) == sort_string(b)
  end

  # Funções privadas
  defp sort_string(string) do
    string
    |> String.downcase()
    |> String.graphemes()
    |> Enum.sort()
  end
end
```

9. Funções anônimas:

Para definir uma função anônima em Elixir nós precisamos das palavras-chave ***fn*** e ***end***. Dentro destes, podemos definir qualquer número de parâmetros e corpos separados por ***->***.



```
defmodule Board do
  defstruct name: nil, items: []

  def remove_item_from_board(%{name: name, items: items}, item_to_remove) do
    new(name, Enum.reject(items, fn item -> item.name == item_to_remove.name end))
  end
end
```

10. Comprehensions:

Comprehensions são um ‘*syntactic sugar*’ (uma forma mais simples de escrever) para realizar loops em ‘**Enum**’ em Elixir.

```
# Módulo math com a função pow para list, elevandp
defmodule Math do
  def pow_2(list) do
    for x <- list, do: x*x
  end
end

iex> list = [1, 2, 3, 4, 5]
iex> Math.pow_2(list)
# [1, 4, 9, 16, 25]
```

11. String:

Strings em Elixir são nada mais que uma sequência de bytes. As strings são representadas como uma sequência de bytes ao invés de um array de caracteres. Elixir também tem um tipo char list (lista de caracteres). Strings em Elixir são delimitadas por aspas duplas, enquanto listas de caracteres são delimitadas por aspas simples.

```
# String

# Sequencia de bytes
iex> string = <<104,101,108,108,111>>
"hello"
iex> string <> <<0>>
<<104, 101, 108, 108, 111, 0>>

# Sequencia de caracteres
iex> 'hello'
[104, 101, 322, 322, 111]
iex> "hello" <> <<0>>
<<104, 101, 197, 130, 197, 130, 111, 0>>
```

12. List

Elixir implementou listas como listas encadeadas. Isso significa que acessar o tamanho da lista é uma operação que rodará em tempo linear ($O(n)$). Por essa razão, é normalmente mais rápido inserir um elemento no início (prepending) do que no final (appending).



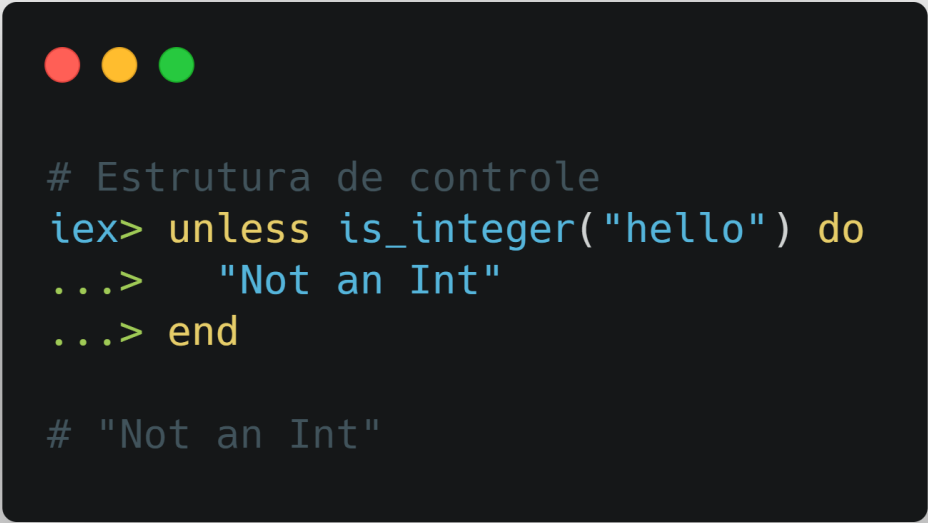
```
# List
iex> list = [3.14, :pie, "Apple"]
[3.14, :pie, "Apple"]

# Prepending (rápido)
iex> ["π" | list]
["π", 3.14, :pie, "Apple"]

# Appending (lento)
iex> list ++ ["Cherry"]
[3.14, :pie, "Apple", "Cherry"]
```

13. Estrutura de controle:

Usar '**unless**' é bem parecido com o uso do '**if**' porém trabalhando de forma negativa.




```
# Estrutura de controle
iex> unless is_integer("hello") do
...>   "Not an Int"
...> end

# "Not an Int"
```

14. Concorrência:

Um dos pontos ofertados pelo Elixir é o suporte a concorrência. Graças à Erlang VM (BEAM), concorrência no Elixir é mais fácil do que esperamos. O modelo de concorrência depende de Atores, um processo contido (isolado) que se comunica com outros processos por meio de passagem de mensagem.



```
# Concorrência
defmodule Example do
  def add(a, b) do
    IO.puts(a + b)
  end
end

iex> Example.add(2, 3)
5
:ok"

iex> spawn(Example, :add, [2, 3])
5
# PID<0.80.0>
```


15. Test unitario:

O framework de testes integrado do Elixir é o *ExUnit*, isto inclui tudo o que precisamos para testar exaustivamente o nosso código. Antes de avançar, é importante notar que testes são implementados como scripts Elixir, por isso precisamos usar a extensão de arquivo **.exs**.

```
defmodule FibonacciTest do
  use ExUnit.Case
  doctest Fibonacci

  test "return fib element 0 correctly" do
    assert Fibonacci.fib(0) == 0
  end

  test "return fib element 1 correctly" do
    assert Fibonacci.fib(1) == 1
  end

  test "return fib element 2 correctly" do
    assert Fibonacci.fib(2) == 1
  end

  test "return fib element 6 correctly" do
    assert Fibonacci.fib(6) == 8
  end

  test "return fib element 7 correctly" do
    assert Fibonacci.fib(7) == 13
  end
end
```

Conclusão

A linguagem Elixir possui características marcantes e interessantes, como podemos perceber a partir deste documento. Como toda linguagem de programação, foi desenvolvida com o intuito de resolver um problema específico, no caso do Elixir ele tinha que resolver o problema de compatibilidade com vários sistemas e ser extremamente escalável. Dessa forma, ele resolveu os objetivos muito bem pois utiliza a máquina virtual do Erlang e ele é totalmente escalável com sua poderosa capacidade de concorrência, podendo criar milhares de processos rapidamente e gerência para comunicar todos os processos via mensagens.

Elixir é usada por empresas como E-MetroTel, Pinterest e Moz. Também é usada para desenvolvimento web, por empresas como Bleacher Report, Discord e Inverse, e para a construção de sistemas embarcados. A comunidade programadora da linguagem, organiza eventos anuais nos Estados Unidos, Europa e Japão, além de eventos e conferências locais menores.

Apresentação visual

Este conteúdo contém uma apresentação visual fornecida pelos autores, disponibilizada na plataforma YouTube <<https://youtu.be/KunCm3ehb0M>>. Os códigos utilizados nesse conteúdo encontram-se disponibilizado no GitHub <<https://github.com/ErickHDdS/elixir>> e <<https://github.com/lucascdornelas/elixir>>