How to clone Virtual RC 2.0 without writing any javascript!

an Elixir/Phoenix/LiveView Worksop

Quick Demos

I wrote zero lines of javascript for these (buggy) prototypes

- Virtual RC Alt
  - https://virtual-rc-alt.gigalixirapp.com
- Pixel Party
  - https://pixel-party.gigalixirapp.com

Pixel Party

We'll be working on a collaborative pixel editor today.

Code and this slide deck: https://github.com/mveytsman/pixel-party

(Virtual RC Alt is at https://github.com/mveytsman/virtual-rc-alt)

Elixir

> Elixir is a functional, concurrent, general-purpose programming language that runs on the Erlang virtual machine (BEAM). Elixir builds on top of Erlang and shares the same abstractions for building distributed, fault-tolerant applications. Elixir also provides productive tooling and an extensible design. The latter is supported by compile-time metaprogramming with macros and polymorphism via protocols.

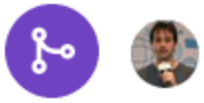
- Heavily inspired by Ruby and Clojure

Ruby

> Ruby is an interpreted, high-level, general-purpose programming language. It was designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan.

Elixir borrows

- Syntax (superficially)
- Focus on programmer ergonomics
- Principle of least surprise
- MINASWAN



josevalim merged commit **698721e** into `phoenixframework:master` 9 hours ago

6 checks passed

josevalim commented 9 hours ago

Clojure

> Clojure is a modern, dynamic, and functional dialect of the Lisp programming language on the Java platform.

Elixir borrows

- Philosophy
- Interop with host VM
- Pipe operator (`(-> ...)`)
- Approach to expression problem

Erlang

> Erlang is a general-purpose, concurrent, functional programming language, and a garbage-collected runtime system. The term Erlang is used interchangeably with Erlang/OTP, or Open Telecom Platform (OTP), which consists of the Erlang runtime system, several ready-to-use components (OTP) mainly written in Erlang, and a set of design principles for Erlang programs.

Erlang

- Active development since 1986
- Originally used for telephony switches by Ericsson
- Kind of looks like Prolog

Erlang Philosophy

- Process-oriented
- Lightweight processes
- No shared state
- Interact through message passing (Actor model)

Elixir

- Functional
- Dynamically typed

OTP in Elixir

- "Let it crash!"
- Supervision trees
- Erlang primitives abstracted for you
  - `GenServer`
  - `Agent`
  - `Task`

Phoenix

> Peace-of-mind from prototype to production
> Build rich, interactive web applications quickly, with less code and fewer moving parts.

- Heavily inspired by Rails
- Familiar MVC-style framework
- Don't need to know any OTP stuff to build APIs or Web apps

Phoenix LiveView

> LiveView provides rich, real-time user experiences with server-rendered HTML.

- New feature of Phoenix
- Server-side templates
- State stored on the server
- Event handlers update state
- LiveView tracks changes sends updates to the browser

Is it stable?

	Version
Erlang	23.0
Elixir	1.10
Phoenix	1.5.3
Phoenix LiveView	0.14.1

Is it stable?

	Version	Status
Erlang	23.0	Older than me
Elixir	1.10	Feature-complete
Phoenix	1.5.3	Stable / battle-tested
Phoenix LiveView	0.14.1	Alpha-quality software

Let's learn elixir!

Elixir the language doesn't have a lot of surface areabut even so, this is a whirlwind tour.

For more see: https://elixir-lang.org/getting-started/

Tooling

- `mix`
  - like `rake`
  - run tasks, generators
- `iex`
  - like `irb` / `pry`
  - REPL
  - for REPL-driven development `iex -S mix ...`

Basic types

- Integers: `123`
- Floats: `9.4`
- Booleans: `true`, `false`
- Strings: `"Hello world"`
- Charlists: `'Hello world'` (beware!)
- Atoms: `:foo`, `Foo`
  - `:foo` - Ruby symbol / Clojure keyword
  - `Foo` - Name of module

Collections

- Tuple: `{1, 2, 3}`
- List: `[1, 2, 3]`
- Maps (dicts): `%{key: 1, key2: 2}`
- Keyword list: `[key: 1, key2: 2]`
  - sugar for `[{:key, 1}, {:key2, 2}]`

Operations

```
iex> [1,2] ++ [2,3]
[1, 2, 2, 3]
iex> [1,2] ++ [3,4]
[1, 2, 3, 4]
iex> [1,2] ++ [3,4]
[1, 2, 3, 4]
iex> "hello" <> " " <> "world"
"hello world"
```

Caveats

- All lists are **linked**-lists
- Keyword-lists and charlists are Erlang holdovers
- When in doubt:
  - `''` for strings
  - `%{}` for maps

Modules & Functions

```
defmodule MyModule do
  def my_function(arg1, arg2)
    private_helper(arg1, arg2)
  end

  defp private_helper(arg1, arg2) do
    arg1 + arg2
  end
end
```

Modules and functions

- All functions live in modules
- `do` / `ends` everywhere
- No explicit `return`
- Everything is an expression (no statements!)

Useful modules

- `Kernel`
- `Enum`
- `String`
- `Map`
  - For maps (dictionaries)! Mapping across a list is `Enum.map`

Functions

Anonymous:

```
iex> fn arg -> arg + 1 end
#Function<7.126501267/1 in :erl_eval.expr/5>
```

```
iex> &(&1 + 1)
#Function<7.126501267/1 in :erl_eval.expr/5>
```

Referring to functions:

```
iex>&Kernel.+/2
&:erlang.+/2
```

Conditionals

- `if`
- `case`
- `cond`

No loops!

But we can `map`

```
iex> l = ["max", "world", "recurse center"]
["max", "world", "recurse center"]
```

```
iex> Enum.map(l, fn x -> "Hello " <> x <> "!" end)
["Hello max!", "Hello world!", "Hello recurse center!"]
```

```
iex> Enum.map(l, &("Hello " <> &1 <> "!"))
["Hello max!", "Hello world!", "Hello recurse center!"]
```

Pipe operator

Passes result as first argument

```
["max", "world", "recurse center"]
|> Enum.map(&("Hello " <> &1 <> "!"))
|> Enum.filter(&(String.length(&1) > 13))
|> List.first

"Hello recurse center!"
```

Comprehensions

Look like loops, but are more like python's generators

```
for x <- ["max", "world", "recurse center"] do
  "Hello " <> x <> "!"
end

["Hello max!", "Hello world!", "Hello recurse center!"]
```

Comprehension generators

```
for x <- [1,2,3], y <- ["a", "b", "c"] do
  {x,y}
end

[
  {1, "a"},
  {1, "b"},
  {1, "c"},
  {2, "a"},
  {2, "b"},
  {2, "c"},
  {3, "a"},
  {3, "b"},
  {3, "c"}
]
```

Comprehension filters

```
for x <- [1,2,3], y <- [1, 2, 3], x != y do
  {x,y}
end

[{1, 2}, {1, 3}, {2, 1}, {2, 3}, {3, 1}, {3, 2}]
```

Pattern matching

`=` is more than asssign!

```
iex> x = 1
1
iex> x
1
```

```
iex(8)> [x,2] = [1,2]
[1, 2]
iex(9)> x
1
```

Pattern matching maps

```
map = %{foo: %{bar: %{baz: 123}, blah: 91}, key: 43}
%{foo: %{bar: bar}} = map

# bar is %{baz: 123}
```

Pattern matching errors

```
iex(10)> [x,3] = [1,2]
** (MatchError) no match of right hand side value: [1, 2]
```

Pattern matching in functions

Consider this `move` function that takes a `position` and a `direction`

```
def move(position, direction) do
  {x,y} = position
  case direction do
    :right -> {x+1, y}
    :left -> {x-1, y}
    :up -> {x, y-1}
    :down -> {x,y+1}
  end
end
```

Pattern matching in functions

We can use pattern matching to get the `{x, y}`

```
def move({x,y}, direction) do
  case direction do
    :right -> {x+1, y}
    :left -> {x-1, y}
    :up -> {x, y-1}
    :down -> {x,y+1}
  end
end
```

If we need to keep the `position` we can write

```
def move({x,y} = position, direction)
```

Pattern matching in functions

We can pattern match on the direction as well!

```
def move({x,y}, :right) do
  {x+1, y}
end
def move({x,y}, :left) do
  {x-1, y}
end
def move({x,y}, :up) do
  {x, y-1}
end
def move({x,y}, :right) do
  {x, y+1}
end
```

Shorthand for `do` / `end`

```
def move({x,y}, :right), do: {x+1, y}
def move({x,y}, :left), do: {x-1, y}
def move({x,y}, :up), do: {x, y-1}
def move({x,y}, :right), do: {x, y+1}
```

Let's make a game!