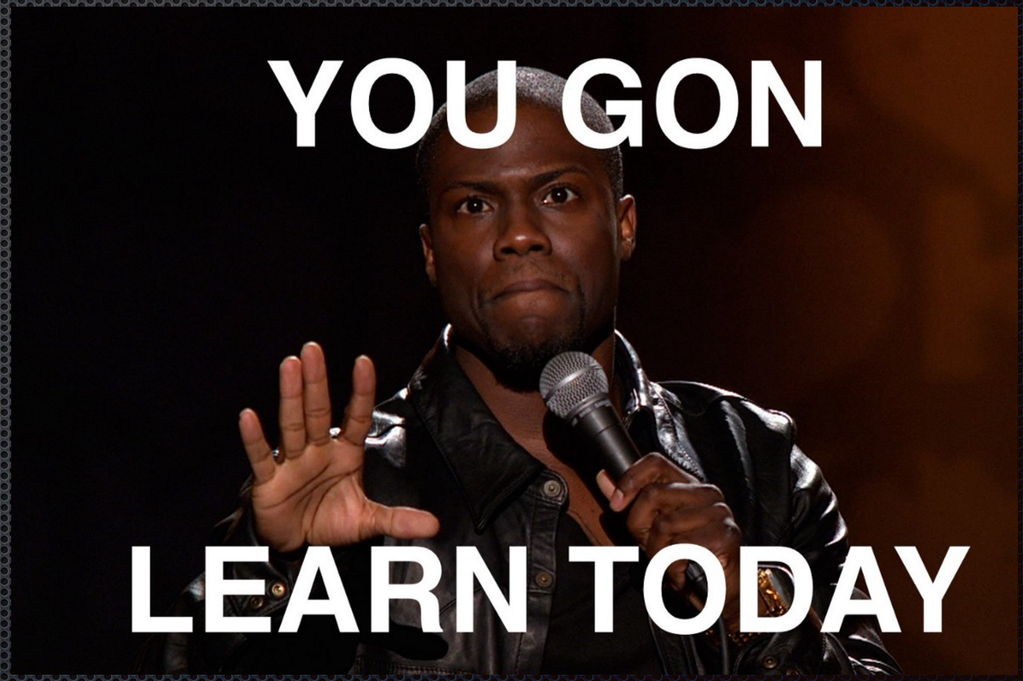
Real-Time User Experiences *with* Elixir and Phoenix LiveView

Dănuț Chindriș

Agenda

- Elixir? What is that?
- Phoenix LiveView
- Feedback

Slack channel: http://bit.ly/peakit003-chat



*10 minutes break every hour

Setting our expectations

- an introduction to Elixir
- first steps with Phoenix LiveView
  - create a small app

Elixir? What is that?

- modern language for today's needs
- crafting beautiful applications functionally

…the future is looking very functional to me. —Uncle Bob

Elixir

- general purpose language
- dynamic language
- promotes functional programming techniques

Who uses Elixir?

- Pinterest
- Discord
- Lonely Planet
- Financial Times
- Heroku

https://elixir-lang.org/cases.html

Make Elixir run your code

in `bin` - three important executable files

- `elixir` - the runtime
- `elixirc` - the compiler
- `iex` - the interactive shell

https://elixir-lang.org/install.html

Elixir files

- scripts: `.exs` files
- run your scripts in IEx with `iex -S scriptname`
- files to be compiled: `.ex` files
- compile them with `elixirc` into `.beam` files and run them with `elixir`
- or compile them in memory and run them directly with `elixir`

Everything is an expression

Data + Functions = ❤️

Data

- just values
- immutable
- values are calculated
- no state to maintain

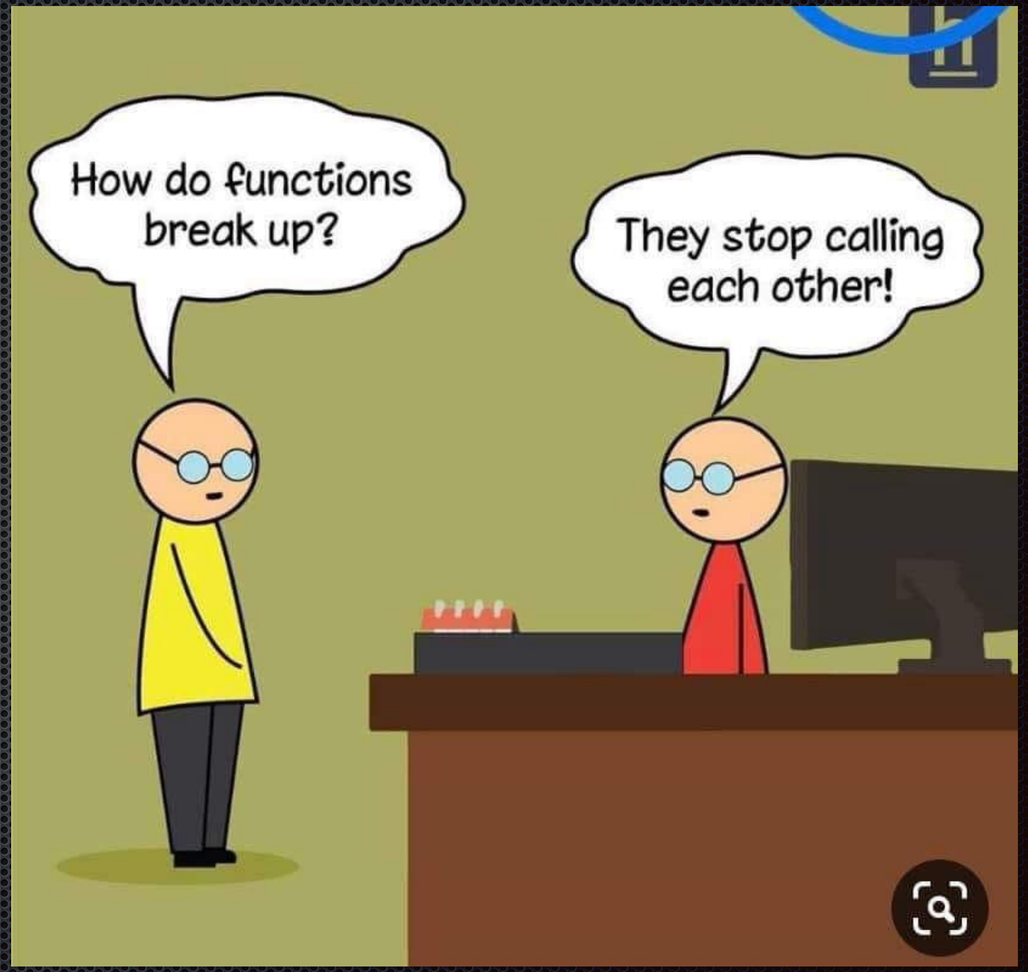
Functions go in modules

- namespaces to organise functions

```
defmodule Flights do
    # your functions go here
end
```

Functions

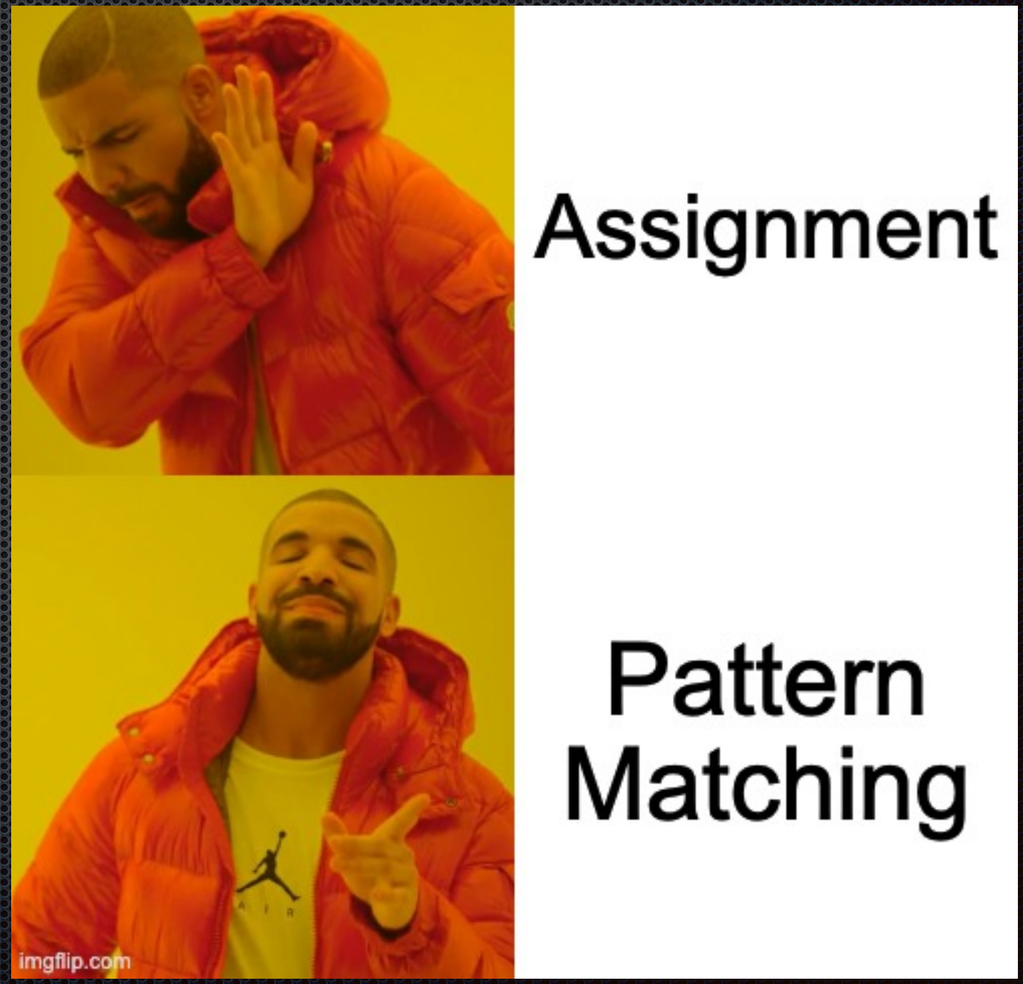
- named functions
- anonymous functions (lambdas)



But first… Pattern Matching

Pattern Matching

- = is called the “match” operator
- not assignment, but “binding”
- variables are labels
- e.g. x = 10 (binds 10 to x)
- e.g. 10 = x (tries to match x against 10)



Pattern Matching

- used for de-structuring
- e.g. `{:ok, result} = {:ok, 1 2 3}`
- e.g. `[h|t] = [“Elixir”, “is”, “awesome”]`

Pattern Matching

- the “pin” operator ^

```
#  binding
x  =  10
#  re-binding
x  =  20
#  match  on  the  existing  value  associated  to  x
^x  =  30
```

Pattern Matching

- _ is called a “wildcard”
- it matches anything
- e.g. `{a,b,_} = {1,2,3}`
- e.g. `{a,b,_c} = {1,2,3}` (c is not bound)

Named functions

- purpose: transform data and return a value
- identified by: name and number of parameters (arity)
  - e.g. `String.to_integer/2`, `Enum.empty?/1`
- documentation: `https://hexdocs.pm/elixir/`

Named functions

```
defmodule Flights do
  def list_airports() do
    [
      {:ARW, "Arad"},
      {:BCM, "Bacău"},
      {:BAY, "Baia Mare"},
      {:OTP, "București"},
      {:CLJ, "Cluj-Napoca"},
      {:CND, "Constanța"},
      {:CRA, "Craiova"},
      {:IAS, "Iași"},
      {:OMR, "Oradea"},
      {:SUJ, "Satu Mare"},
      {:SBZ, "Sibiu"},
      {:SCV, "Suceava"},
      {:TGM, "Târgu Mureș"},
      {:TSR, "Timișoara"},
      {:TCE, "Tulcea"}
    ]
  end
end
```

Named functions

- `def` - exported function
- `defp` - private function (not exported)
- convention: snake_case for function names
- the value of the last expression is the returned value

Function clauses

- we use pattern matching to separate a function's logical branches
- the first function clause that matches is executed

```
defmodule Lists do

  def head([]) do
    {:error, "empty list"}
  end

  def head([h | _t]) do
    {:ok, h}
  end

end
```

Anonymous functions (lambdas)

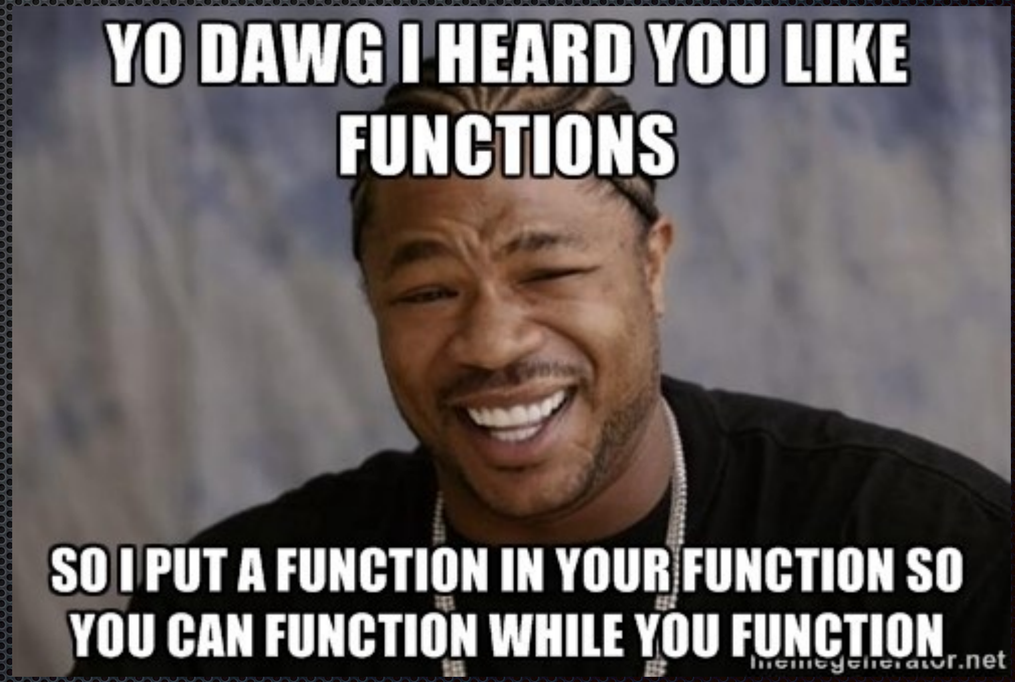
```
fn {_iata, city} -> city end
```

Using the capture operator
(please don’t freak out 😅)

```
&(elem(&1, 1))
```

Higher-order functions

- functions which take other functions as parameters and/or return functions
- *functions as data*
- lambdas and functions defined with “capture” are useful for HOFs

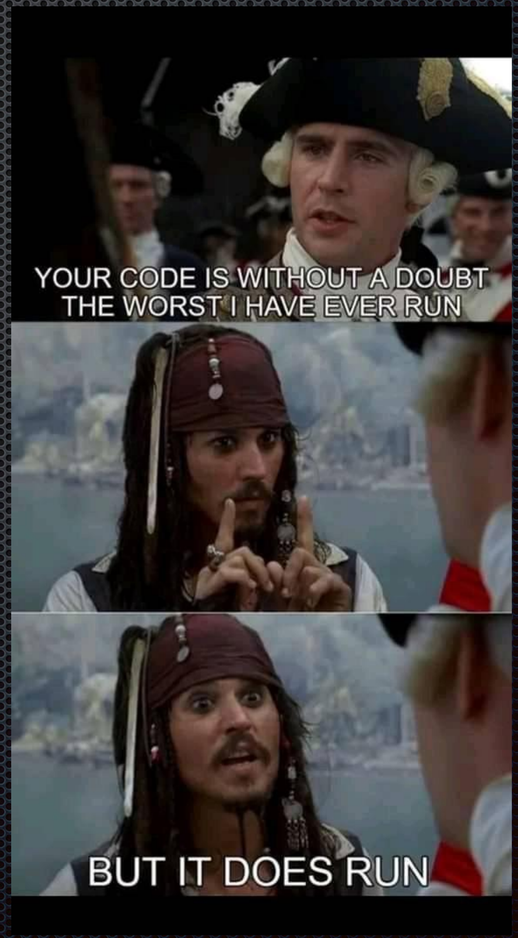


Functional patterns

- FP supports high-level patterns such as *map*, *filter*, *reduce*
- implemented with HOFs
- frameworks such as Hadoop have adopted functional patterns

Why use functional patterns?

- reuse: define “map” once, reuse it whenever we want to transform elements
- comprehensibility: we see “map”, we know what’s going on



map

- when you see “map” think transforming the elements of an enumerable
- `Enum.map/2`
- e.g. add 1 to each element of a list containing numbers
  - `Enum.map([1,2,3], fn x -> x+1 end)`

filter

- when you see “filter” think selecting just the elements which respect a set of conditions (predicate)
- `Enum.filter/2`
- e.g. take the even elements in a list containing numbers
  - `Enum.filter([1,2,3,4], fn x->rem(x,2)==0 end)`

reduce

- when you see “reduce” think smooshing the values of an enumerable together
- `Enum.reduce/3`
- e.g. sum the elements of a list
  - `Enum.reduce([1,2,3],0, fn x,acc -> x+acc end)`

The “pipe” operator |>

```
def find_city_by_code(code) do
  {_iata, city} =
  list_airports()
  |> Enum.find(fn {iata, _city} -> iata == code end)

  city
end
```

Phoenix

- MVC (fresh with functional ideas)
- Real-Time (Channels & Presence)
- Versatility (through Erlang VM)
- Production Ready (built-in instrumentation, live dashboard)

Phoenix

```
connection
#  functions  which  are  called  for  each  request
|>  endpoint()
|>  router()
#  the  connection  goes  through  a  named  pipeline
#  which  has  common  functions  for  each
#  main  request  type
|>  pipeline()
#  it  invokes  the  model  and  renders  a  template
#  through  a  view
|>  controller()
```

Real-time applications

- using Channels and Presence technologies:
  - interact with users, push notifications to multiple clients
  - know who is connected right now, across many nodes
- using LiveView:
  - real-time apps without the client-side complexities

LiveView

- real-time user experiences with server-rendered HTML
- no need to manage both client and server
- LiveView automatically updates the client

LiveView

- first renders statically as part of regular HTTP requests
- then creates a persistent WebSocket connection
- it reacts faster to user events
  - less work to do
  - less data to send

LiveView

- basically:
  - great for backend developers
  - write Elixir code and get both client and server managed
- example: http://localhost:4000/dashboard/

Useful commands

- create project: `mix phx.new project_name --live`
- start app: `mix phx.server`
- get live info: `window.liveSocket.enableDebug()`

Programming model

- create Elixir modules
- write a few functions:
  - `mount/3`
  - `render/1`
  - `handle_event/3`
  - `handle_info/2`
- add routes to `router.ex`

Live coding - LiveView application

Thank you!

It’s ⏳ to ⚗️ some Elixir 🧪

Feedback*

https://bitly.com/peakit003-feedback

*pretty please 😁