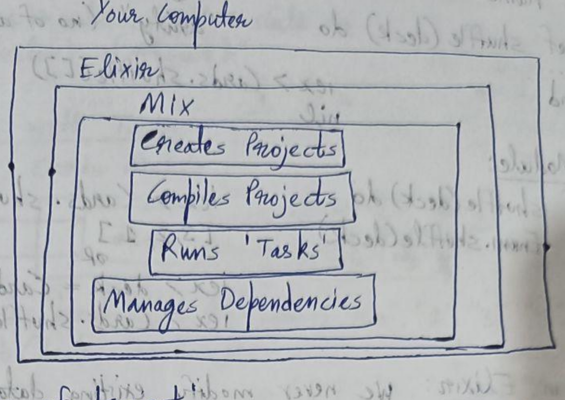
# (The Complete Elixir and Phoenix Bootcamp)

## Generating a Project:



```
mix new name_of_project
```

## Elixir Modules and Methods:

Nearly all the code that we write in Elixir is organized in modules. A module is a collection of different methods or functions.

```
defmodule Cards do
    def hello do
        "hi there!"
    end
end
```

$\downarrow$ implicit return: last value gets returned anyways

Run the program using:

```
iex -S mix
```

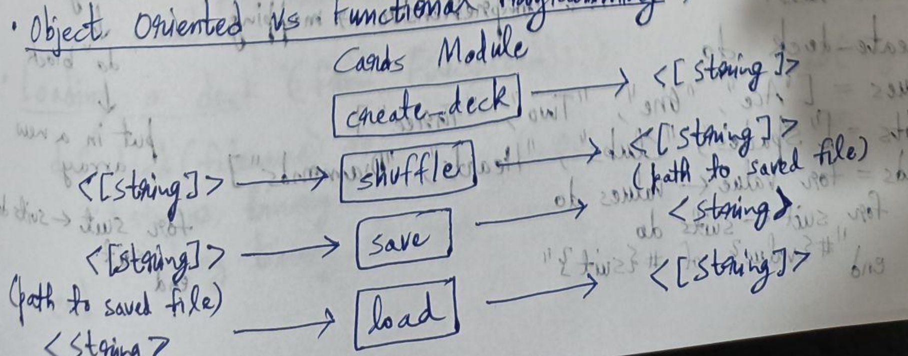
$\hookrightarrow$ interactive elixir shell

## Lists and Strings:

```
defmodule Cards do
    def create_deck do
        ["Ace", "Two", "Three"]
    end
end
```

To recompile: recompile while the shell is running

## Object Oriented vs Functional Programming:



• Error handling:     atoms are used throughout our application to handle status_codes. Think of atom like a string.

```
def load(filename) do
  {status, binary} = File.read(filename)
  case status do
atom  :ok -> :erlang.binary_to_term binary
      :error -> "That file does not exist"
  end
end
```

• Pattern matching in case statements:

```
# ["red", color] = ["red", "blue"] → color gets assigned to "blue".
# ["red", color] = ["green", "blue"] → will throw an error.
```

Simultaneously do comparison and assignment:-

```
def load(filename) do
  case File.read(filename) do
    {:ok, binary} -> :erlang.binary_to_term binary
    {:error, _reason} -> "That filename does not exist"
  end    ↳ for variables that we don't want to use
end
```

• The pipe operator:

$$\begin{tikzcd}
\boxed{\text{Cards.create\_deck}} \arrow[r] & \boxed{\text{List of cards}} \arrow[d] & & \\
 & \boxed{\text{Cards shuffle}} \arrow[r] & \boxed{\text{List of Cards}} \arrow[d] & \boxed{\text{Hand size}} \arrow[dl] \\
 & & \boxed{\text{Cards.deal}} \arrow[r] & \boxed{\text{List of cards}}
\end{tikzcd}$$

Pipe operator can be used to setup the chain of method calls.

Without pipe operator

```
def create_hand(hand_size) do
  deck = Cards.create_deck
  deck = Cards.shuffle(deck)
  hand = Cards.deal(deck, hand_size)
end
```

with pipe operator

```
def create_hand(hand_size) do
  Cards.create_deck
  |> Cards.shuffle
  |> Cards.deal(hand_size)
end
```

automatic injection of arguments

• Module documentation:

① manually write inside mix.exs.

```
defp deps do
  [
    {:ex_doc, "0.12"}
  ]
end
```

↓ name of package   ~> ↓ version

② Run `mix deps.get`

There are 2 types of documentation that we write with x-doc:
• Module documentation   • Function documentation

```
@doc """
<write function doc here>
"""
```

(i). Module doc:

```
defmodule Cards do
@moduledoc """
<write documentation here>
"""
```

with example code.

ex.,

```
@doc """
Divides the deck into a hand and remainder of the deck. The `hand_size`
argument determines the size of the hand.

## Examples
<space>
  → 3 tab chars
    iex> deck = Cards.create_deck
    iex> {hand, deck} = Cards.deal(deck, 1)
    iex> hand
    ["Ace of Spades"]
```

← Doc For deal fxn
docs/index.html

• Introduction to Testing:   run `mix test`

There are 2 tests: tests for finality and test for checking Examples section inside comments.

```
defmodule CardsTest do
  use ExUnit.Case
  doctest Cards    # enables checking of code in Examples section (Doctest)
end
```

• Case Tests:

```
defmodule CardsTest do
  use ExUnit.Case
  doctest Cards
  test "create_deck makes 20 cards" do
    deck_length = length(Cards.create_deck)
    assert deck_length == 20
  end
end
```

• Maps:

```
iex> colors = %{primary: "red", secondary: "blue"}
iex> colors.secondary
"blue"
iex> %{secondary: secondary_color} = colors
iex> secondary_color
"blue"
```

• Updating values in a map: (1). Using function:

```
iex> Map.put(colors, :primary, "blue")
%{primary: "blue"}
```

• **Method Arguments**: In Elixir, we can have more than one method with the same name (shuffle/0 takes 0 args, shuffle/1 takes 1 args)
↓
"arity" (no of args.)

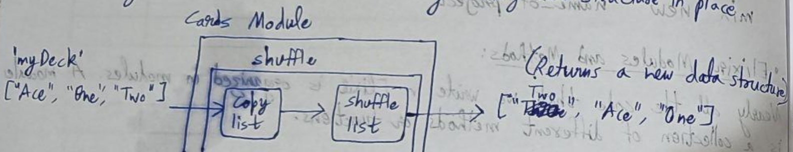
```
def shuffle (deck) do
...
end
```

• **The Enum Module**:

```
def shuffle(deck) do
  Enum.shuffle(deck)
end
```

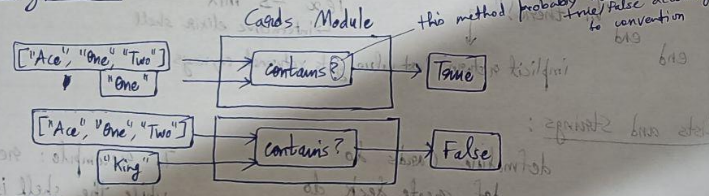
```
iex > Cards.shuffle([])
nil

iex > Cards.shuffle([1,2,3])
[3,2,1]
        OR
iex > deck = Cards.create_deck()
iex > Cards.shuffle(deck)
```

• **Immutability in Elixir**: We never modify existing data structure in place.



(Returns a new data structure)

• **Searching a list**:



this method probably returns true/false according to convention

```
def contains?(deck, card) do
  Enum.member?(deck, card)
end
```

• **Comprehension over lists**: list comprehension → mapping

```
def create_deck do
  values = ["Ace", "One", "Two", "Three"]
  suits = ["Spades", "Clubs", "Hearts", "Diamonds"]
  cards = for value ← values do
    for suit ← suits do
      "#{value} of #{suit}"
    end
  end
```

array
↓
"do" block
↓
put in a new array

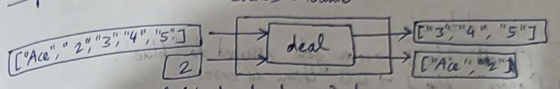
```
for suit ← suits do
  suit
end
```

```
  List.flatten(cards)      // Flattens lists of lists to a single list
end
```

OR better ✓

```
~~cards~~ = for value ← values, suit ← suits do
  "#{value} of #{suit}"
end
```

• **Importance of Index with Tuples**:



```
def deal (deck, hand_size) do
  Enum.split(deck, hand_size)
end
```

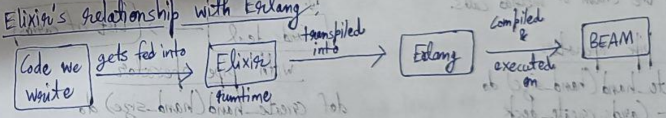
cards = ["Ace", "2", "3", "4", "5"]
↓ Enum.split(cards, 2)
Tuple → {["3", "4", "5"], ["Ace", "2"]}

Enum.split(deck, hand_size) ⟶ { $\boxed{\text{My hand}}$ , $\boxed{\text{The Rest}}$ }
My hand is always at index 0 ↑ ↑ Rest of deck is always at index 1

***Pattern matching**: is Elixir's replacement for variable assignment.

```
{hand, rest_of_deck} = Cards.deal(deck, 5)
  # iex > hand   will print hand since it stored reference to hand and vv.
# iex > [color1] = ["red"]   means color1 = "red"
# iex > [color1, color2] = ["red", "blue"]   means color1 = "red" & color2 = "blue".
# iex > [color1, color2, color3] = ["red", "blue"] → error
```

• **Elixir's relationship with Erlang**:



• **Saving a deck** (to FileSystem):

```
def save(deck, filename) do
  binary = :erlang.term_to_binary(deck)   #encode the file
  File.write(filename, binary)            # Create a file named "filename"
end
```

Cards.save(deck, 'my_deck')

• **Loading a deck** (from FileSystem):

Cards.load('my_deck')

```
def load(filename) do
  {status, binary} = File.read(filename)
  :erlang.binary_to_term binary
end
```

(ii). Using Elixir syntax:

```
iex> colors
%{primary: "red"}          → we never change existing ds
iex> %{colors | primary: "blue"}   # can only be used
```

if we are updating an existing property. It cannot be used if we want to add a new property to map. But fxns can be used Map.put(colors, :secondary, "green")

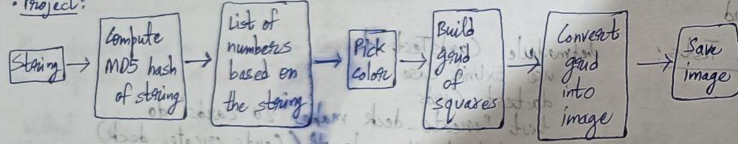
```
iex> "
%{primary: "red", secondary: "green"}
```

• Keyword lists: list + tuple

```
iex> colors = [{:primary, "red"}, {:secondary, "green"}]
[primary: "red", secondary: "green"]
iex> colors[:primary]
"red"
iex> colors = [primary: "red", secondary: "blue"]
[primary: "red", secondary: "blue"]
iex> colors = %{primary: "red", primary: "blue"}
%{primary: "blue"}
iex> colors = [primary: "red", primary: "red"]
[primary: "red", primary: "red"]
```

If we are passing keyword list as the last parameter, then square brackets can also be ignored

• Project:



• Structs: are used to model data in Elixir. A struct is a map used to store data in an Elixir application. They (structs) have 2 advantages over maps: 1. They can be assigned default values, 2. Compile-time checking of properties.

```
defmodule Identicon.Image do
    defstruct hex: nil  #stores list of numbers
end
iex> %Identicon.Image{}
%Identicon.Image{hex: nil}
iex> %Identicon.Image{hex: []}
%Identicon.Image{hex: []}
```

Why to use structs instead of map?
- Structs enforce that the only property that exist on the struct are the ones defined in the module file

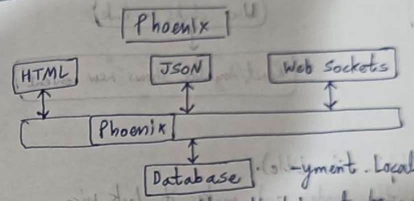
• Pattern matching structs:

```
%Identicon.Image{
    hex: [40, 24, 2, 190, ...]
}
%Identicon.Image{hex: hex_list} = image
```

Ist Three elements `%Identicon.Image{hex: [a, b, c | _tail]} = image`

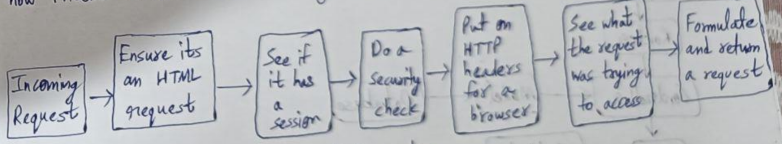
• Adding element to the end of list: `row ++ [second, first]`
`[first, second | _tail] = row`

• Mirror rows: `Enum.map(&mirror_row/1)` → arity = 1; 1 args

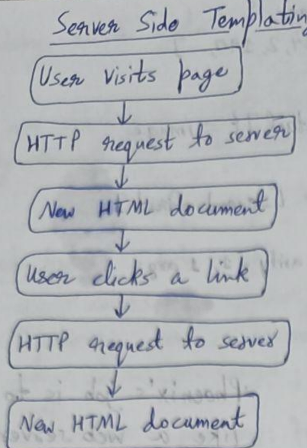


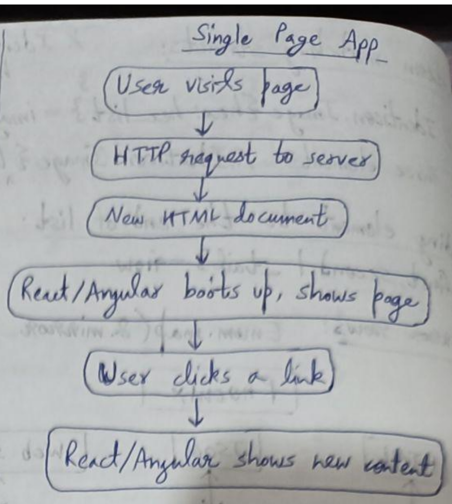
Phoenix's job is to behave like a web server. So, its going to be hosted on some remote server in deployment. Locally, we're going to host it on our computer during the development process When we deploy a Phoenix server, users can interface with the server and receive back HTML

The most important aspect of Phoenix is to understand how it handles these incoming requests. Once, you kind of internalize or really understand how Phoenix handles these incoming requests, you will be a master of Phoenix.


Incoming Request
Ensure its an HTML request
See if it has a session
Do a Security check
Put on HTTP headers for a browser
See what the request was trying to access
Formulate and return a request

```
mix phoenix.new <name_of_project>
mix ecto.create
mix phoenix.server
```

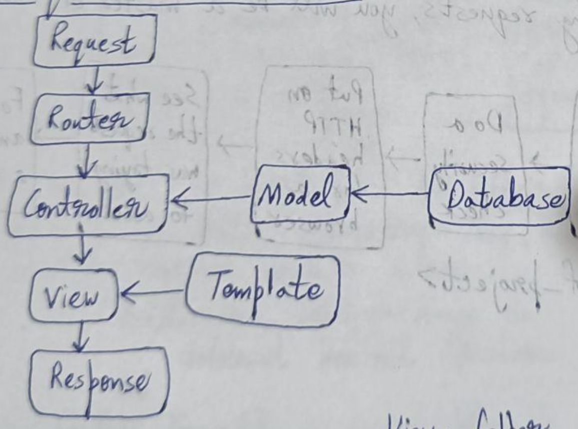




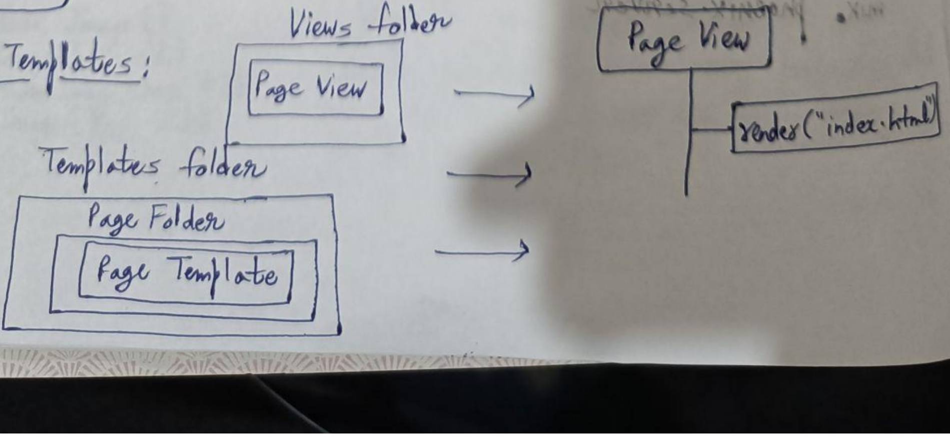
• Phoenix's MVC model:

Model: The raw data of the topic. (its title).
View: A template that takes the model and makes it look nice.
Controller: Figures out what the user is looking for, grabs the correct model, stuffs it into the view, returns the result to the user.

• How requests work in Phoenix:



• Views vs Templates:



When Phoenix
that views f
the templates
There is an
we have insi
to the nam
templates f
a fxn calle

So, if we c
return to u

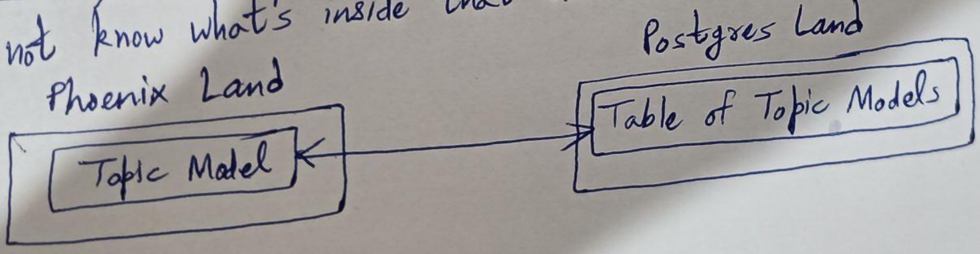
• Model l

does not

When Phoenix first boots up, it looks in our views folder. For every module in that views folder, it takes the name of model. Phoenix will then look inside the templates folder and will try to find a folder with name 'Page' in it.
There is an intrinsic link b/w the name of our views and the folders that we have inside templates folder. Because, the name of the folder matches up to the name of the view, Phoenix is going to take every file inside of the templates folder and its going to add it as a fxn to the page view, specifically a fxn called render with the argument index.html.

So, if we call PageView.render and pass in a string index.html, it's gonna return to us a template of index.html

• <u>Model layer in Phoenix</u> : Phoenix knows that there is a database but it does not know what's inside that database


Phoenix Land
Topic Model
Postgres Land
Table of Topic Models