

Elixir Workshop

Julius Putra Tanu Setiaji

13 May 2019

Where are we?

First Steps

Erlang

Elixir

Source Academy

Preparing Environment

Building Blocks

Control Flow

Other Useful Features

About Erlang



- A development platform for building **scalable** and **reliable** systems.
- Built in Ericsson¹ in the 1990s
- Runs on BEAM (Björn's Erlang Abstract Machine)².

¹Ericsson was and is one of the largest telecom infrastructure companies in the world.

²Similar idea: Java runs on the JVM (Java Virtual Machine)

High Availability

- Fault tolerance
- Scalability
- Distribution
- Responsiveness
- Live update

A tribute to Prof. Joe Armstrong

Prof. Joe Armstrong was behind the development of Erlang. He passed away recently on 20th April 2019.



Sample Erlang

Based on **Prolog**: its syntax borrows heavily from it and the first Erlang compiler³ was written in it too.

```
-module(fact).  
-export([fac/1]).
```

```
fac(0) -> 1;  
fac(N) when N > 0, is_integer(N) -> N * fac(N - 1).
```

³https://www.erlang.se/publications/prac_appl_prolog.ps

Where are we?

First Steps

Erlang

Elixir

Source Academy

Preparing Environment

Building Blocks

Control Flow

Other Useful Features

About Elixir



- Targets BEAM, with full Erlang interoperability⁴.
- Started by José Valim in 2012 – he was involved heavily in the Ruby on Rails coreteam before.
- Provides almost one-to-one mapping to Erlang constructs but with additions to reduce boilerplate and duplication.

⁴It means that we can use Erlang libraries and tooling

Sample Elixir

The syntax is heavily borrowed from Ruby.

```
defmodule Fact do
  def fac(0) do
    1
  end

  def fac(n) when n > 0, is_integer(n) do
    n * fac(n - 1)
  end
end
```

Phoenix Framework



- Elixir's web framework, just like RoR in Ruby, or Django in Python.
- Initially it was heavily based on Ruby on Rails.
- Over time, Phoenix has diverged from its Rails roots and developed its own unique ideas.

Ecto



- A database wrapper and language integrated query for Elixir.
- Data mapping and validation, with a SQL adapter.
- Conceptually, this is the Model in MVC architecture. Similar to ActiveRecord in Ruby on Rails.

Where are we?

First Steps

Erlang

Elixir

Source Academy

Preparing Environment

Building Blocks

Control Flow

Other Useful Features

Roles of Elixir in Source Academy

- Source Academy has a separate **backend** and **frontend**.
- **Backend** stores data and does the business logic.
- **Frontend** is what the user interacts with and makes requests to the backend.
- Phoenix Framework is used to write the Source Academy's backend.

Why Elixir?

- Elixir has good performance due to BEAM (compared to the alternatives, such as Ruby on Rails, Express on node.js, etc.).
- Elixir is a functional language, in line with what CS1101S and SICP taught.
- Elixir has more familiar syntax than Erlang.
- Good package manager (**hex** and **rebar**), which provides ability to use Erlang and Elixir libraries.

Where are we?

First Steps

Erlang

Elixir

Source Academy

Preparing Environment

Building Blocks

Control Flow

Other Useful Features

Steps of Installing Elixir

1. **Install Erlang:** I prefer using the native package manager for this (whatever the latest OTP version is)
2. **Install Elixir:** I prefer using **asdf** so I can manage more than 1 version of Elixir at the same time

Installing Erlang

On Mac: https://is.gd/install_erlang_mac

On Ubuntu:

https://is.gd/install_erlang_ubuntu

Check if you have a working Erlang/OTP 21 installation:

```
$ erl
```

```
Erlang/OTP 21 ...
```

```
Eshell ...
```

```
> io:fwrite("Hello, world!~n")
```

```
Hello, world!
```

```
ok
```

Installing asdf and Elixir

Go to <https://asdf-vm.com/>, click on “Get Started” and follow the instructions. Afterwards:

```
asdf plugin-add elixir
asdf install elixir 1.8.1-otp-21
```

Check if you have a working Elixir 1.8.1 installation:

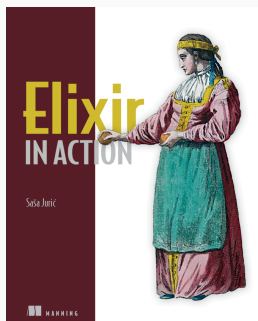
```
$ iex
Erlang/OTP 21 ...
```

```
Interactive Elixir (1.8.1) ...
```

```
iex(1)> IO.puts("Hello, world!")
Hello, world!
:ok
```

Recommended Reading

Elixir in Action by Sasa Juric



The first part would do (first 4 chapters)

Where are we?

First Steps

Building Blocks

The Interactive Shell

Variables

Organising Your Code

Types

Control Flow

Other Useful Features

Elixir Interactive Shell (REPL)

- You can enter Elixir's REPL⁵ by running `iex` from terminal.
- `i`: inspect a value. Example: `i 5`
- `h`: get help about a command. Examples:
 - h `Enum.reduce`
 - h `Enum.reduce/2`
 - h `Enum.reduce/3`
- Note that in Elixir, everything is an expression and thus will return something.

⁵Read-Evaluate-Print-Loop

Where are we?

First Steps

Building Blocks

The Interactive Shell

Variables

Organising Your Code

Types

Control Flow

Other Useful Features

Variables

- Elixir is a *dynamic* language, thus there is no explicit type declaration.
- In Elixir, mapping a value to a variable is called **binding**.
- To do this, we use the match operator =
- Variable names uses snake_case style.
- Example:

```
x = 1  
x  
y = "a"  
y  
x = 2  
x
```

Immutability

- Elixir is a functional language where everything is immutable.
- Re-binding a variable changes the value that a variable is pointing to but never the value itself.
- Let's test it with anonymous function:

```
x = 42  
foo = fn -> IO.puts(x) end  
x = 0  
IO.puts(x)  
foo.()
```


Where are we?

First Steps

Building Blocks

The Interactive Shell

Variables

Organising Your Code

Types

Control Flow

Other Useful Features

Organising Your Code

- As a functional language, Elixir relies heavily on functions.
- Due to immutable nature of data, typical Elixir program consists of many small functions.
- Multiple functions are grouped together into modules.

Module and Functions

- A collection of functions – similar to namespace in other languages.
- Every named function in Elixir must be defined inside a module.
- Syntax: `ModuleName.function_name(args)`
- Module names use CamelCase style. It can also contain the dot character to organise modules hierarchically.
- Module definition starts with the `defmodule` construct, followed with a `do-end` block.
- Nested module or module whose names contains dot has no special relation – all modules are independent of one another (for now).

Module and Functions (cont.)

- Function names use snake_case style.
- Function name conventions: ? suffix means the function returns a boolean, ! suffix means the function may raise a runtime error.
- Function definition starts with the **def** construct, followed with a **do-end** block.
- Parentheses can be omitted in definition of 0-arity functions.
- No explicit return value – instead, last expression is the return value.

Modules and Functions (cont.)

- Default argument is declared using `\` – this creates functions with the same name but different arities
- In general, ignored variable are prefixed with underscores.

Module and Functions (cont.)

Save as `Geometry.ex`, run as `iex Geometry.ex`:

```
defmodule Geometry do
  def foo, do: "Hello, world!"

  defmodule Square do
    def area(side \\ 2), do:
      ↪ Geometry.Rectangle.area(side, side)
  end
end

defmodule Geometry.Rectangle do
  def area(length, width) do
    length * width
  end
end
```

Module and Functions (cont.)

```
Geometry.Rectangle.area(2, 5)
```

```
Geometry.Square.area(3)
```

```
Geometry.foo()
```

Function visibility

- Functions defined using `def` is public (exported) and can be used by any module.
- To make a function private, define using `defp` – private functions can only be invoked inside the module where it is defined.

Imports and Aliases

- **import** allows calling public functions of a module without prefixing with the module name.
- **alias** allows referencing a module under a different name.
- Example:

```
import IO
puts("Hello, world!")
alias Geometry.Square
Square.area(3)
alias Geometry.Square, as: MySquare
MySquare.area(3)
```

Module Attributes

Has 3 uses:

- As annotations
- As constants at run-time
- As temporary storage at compile-time

Module Attributes: As annotation

Save as `Foo.ex`, then compile by running `elixirc Foo.ex`, then run `iex`

```
defmodule Foo do
  @moduledoc """
    This is the documentation for the Foo module.
    """

  @doc "bar/0 returns the number 5"
  def bar, do: 5
end
```

Module Attributes: As constants at run-time

```
defmodule Geometry.Circle
  @pi 3.14159

  def area(radius), do: @pi * radius * radius
end
```

Comments

- Comments start with the character `#`
- Block comments are not supported – instead prefix each one with `#`

Where are we?

First Steps

Building Blocks

The Interactive Shell

Variables

Organising Your Code

Types

Control Flow

Other Useful Features

Identifying functions

- Functions in Elixir are identified by their name and arity.
- The arity of a function describes the number of arguments that the function takes.
- Example: `Enum.reduce/2` identifies a function from the `Enum` module, function name is `reduce`, and the arity is 2, while `Enum.reduce/3` describes the same module, the same function name, but with arity 3.

Integer and Float

- In Elixir, just like in Erlang and Ruby, Integer has arbitrary precision, while float has 64-bit double precision.
- Predicate functions: `is_integer/1` and `is_float/1`
- Note that the `/` operator always return a float.
- To get integer division, use `div/1` instead.
- To get remainder, use `rem/1`

Integer and Float (cont.)

- Examples:

```
1 + 2
```

```
5 * 5
```

```
10 / 2
```

```
div(10, 2)
```

```
rem(10, 3)
```

```
rem 10, 3
```

- Note that in Elixir, you can drop parantheses when invoking named functions, just like Ruby (generally discouraged)

Integer and Float (cont.)

- Elixir also provides shortcut notation to enter binary, octal, and hexadecimal numbers:

```
0b1010
```

```
0o777
```

```
0x1F
```

- Float requires a dot followed by at least 1 digit.
- It also supports `e` for scientific notation.
- You can use `floor/1`, `ceil/1`, `trunc/1`, `round/1`.

```
1.0
```

```
1.0e-10
```

```
round(-1.5)
```

```
trunc(-1.5)
```

```
floor(-1.5)
```

```
ceil(-1.5)
```

Boolean

- Either the value `true` or `false`
- Predicate function: `is_boolean/1`
- Examples:
 `is_boolean(true)`
 `is_boolean(false)`
 `is_boolean(5)`

Atom

- A constant whose name is its own value. Similar to Symbols in Ruby or Lisp.
- Comparison is $O(1)$, while keeping the value named instead of an integer constant.
- Syntactically, written with a colon `:` prefix (like Ruby)
- Predicate function: `is_atom/1`
- Example:
 `:test`
 `is_atom(:test)`

Atom (cont.)

- Note that in Elixir (and Erlang), booleans are implemented as atoms `:true` and `:false`, and `nil` as `:nil` too.
- Module names are atoms too.
- You can also specify atom name that contains special characters (such as dot or colon) by delimiting them with double-quotes.

```
is_atom(true)
is_atom(Tuple)
:"Elixir.Tuple"
:"asd:a.sdd"
```

String

- Delimited by double quotes, encoded in UTF-8
- Represented internally by binaries (sequence of bytes). Binaries are delimited with << and >>
- Predicate function: `is_binary/1`
- Examples:

```
"Hello, world!"
```

```
<<104, 101, 108, 108, 111>>
```

```
is_binary("Hi!")
```

- Elixir supports string interpolation:

```
"Hello, #{:world}"
```

String (cont.)

- Get number of bytes in a string using `byte_size/1`
- Get length of string using `String.length/1`
- However, they might not be the same as the length of the string due to UTF-8 encoding:

```
byte_size("hellö")  
String.length("hellö")
```

- `String` module contains helpful functions to manipulate string:

```
String.upcase("hellö")
```

Anonymous functions (lambda)

- Delimited by keywords **fn** and **end**
- Functions are first-class citizens: they can be passed as arguments to other functions.
- Note that a dot between the variable and parantheses is required to invoke an anonymous function.
- Predicate functions: `is_function/1`, `is_function/2`
- Examples:

```
add = fn a, b -> a + b end
```

```
add.(1, 2)
```

```
is_function(add)
```

```
is_function(add, 2)
```

```
is_function(add, 1)
```

```
Enum.each('hello', fn x -> IO.puts(x) end)
```


The Capture Operator

- Ampersand & is the capture operator.
- It can be used to create anonymous function
- Examples:

```
Enum.each('hello', &IO.puts/1)  
Enum.each('hello', &IO.puts(&1))  
(&IO.puts(&1)) == &IO.puts(&1)
```

(Linked) Lists

- Dynamic, variable-sized collections of data.
- The syntax might look like an array, but it actually is a linked list with $O(n)$ complexity for most functions.
- It uses Lisp-y list (SICP list): the lists are built from pairs
- Syntax for pair: `[head | tail]`
- To get head and tail, use `hd/1` and `tl/1` respectively.
- Predicate functions: `is_list/1`

(Linked) Lists

■ Examples:

```
[1 | 2]
```

```
[1 | [2 | []]]
```

```
[1, 2]
```

```
is_list([1, 2, 3, 4])
```

```
length([1, 2, 3, 4])
```

■ Concatenate using the ++/2 operator, subtract using the --/2 operator:

```
[1, 2, 3] ++ [4, 5, 6]
```

```
[1, true, 2, false, 3, true] -- [true, false]
```

(Linked) Lists (cont.)

- Note that in Erlang, strings are usually represented as charlist (list of characters) instead of binaries.
- Charlist is written in Elixir delimited single quote.
- Examples:

```
'hello'
```

```
[104, 101, 108, 108, 111]
```

```
"hello"
```

```
<<104, 101, 108, 108, 111>>
```

```
'hello' == "hello"
```

Tuple

- Delimited by curly braces.
- Group a fixed number of elements, stored contiguously in memory. Thus, most operations are $O(1)$

- Examples:

```
tuple = { :ok, "world" }  
tuple_size({ :ok, "world" })  
elem(tuple, 1)  
put_elem(tuple, 1, "world")
```

List vs Tuple

- Appending lists is $O(n)$
- Tuples are stored contiguously in memory, thus updating a value in tuple is expensive as a new tuple has to be created.
- Tuples are typically used to return more than 1 data from a function: `{:ok, data}`, `{:error, :reason}`
- Usually, Elixir will guide you to do the right thing: `elem/1` exists but no built-in equivalent for lists.
- When counting elements, in Elixir, `size` signifies $O(1)$, while `length` signifies $O(n)$
- E.g. `byte_size/1`, `tuple_size/1` vs `length/1`, `String.length/1`

Operators

- **Arithmetic:** `+`, `-`, `*`, `/`, `div`, `rem`
- **List:** `++`, `--`
- **Binary (String):** concatenate `<>`
- **Boolean:** `and`, `or`, `not`
- **Truthy/Falsey:** `||` return the first truthy value or the last element, `&&` return the first falsey value or the last element, `!` returns `true` except for `false` and `nil`
- **Comparison:** `==`, `!=`, `<=`, `>=`, `>`, `<`, `===` strict compare integer and float, `!==`
- Different types can be compared with total order⁶.

⁶`number < atom < reference < function < port < pid < tuple < map < list < bitstring`

Maps

- A key-value store, implemented using Hash Array Mapped Trie (HAMT).
- Examples:

```
%{a: 1, b: 2}
```

```
%{:a => 1, :b => 2}
```

```
map = %{ "a" => %{ "b" => [true, false, nil] }, 5  
  ↪    => "boo", :z => 100 }
```

```
map[ "a" ]
```

```
map[ "a" ][ "b" ]
```

```
map[ :z ]
```

```
map.z
```

```
%{map | 5 => "honhonhon" }
```


Keyword List

- Older way to create a key-value store is having a list of 2-item tuple.
- If the first item of the tuple is an atom, then this is a keyword list.
- Example:
`[{:a, 1}, {:b, 2}]`
`[a: 1, b: 2]`
- Important properties:
 - Keys must be atoms
 - Keys are ordered
 - Keys can be given more than once.
- Beware of the $O(n)$ performance characteristics.

Range

- Represents a range of numbers (like Ruby)
- Example:

```
range = 1..2
```

```
1 in range
```

```
-1 in range
```

```
Enum.each(1..3, &IO.puts/1)
```

Macros

- Advanced feature – we will not go into much detail.
- A very lisp-y feature.
- Elixir metaprogramming feature: code that receives AST and manipulates them.
- Note that many of the constructs in Elixir are actually implemented as macros on the standard library: `def`, `defp`, etc.

Where are we?

First Steps

Building Blocks

Control Flow

Pattern Matching

Conditionals

Recursion

Enumerable

Streams

Comprehensions

Pattern Matching

- One of the most powerful features of functional languages.
- Similar to destructuring in Ruby or JavaScript.
- Recall the match operator =
- So far we have just done simple bindings from the RHS to LHS:

`x = 1`

`x`

Pattern Matching (cont.)

- However, we can also do:
 $1 = x$
- This is because LHS and RHS are both 1.
- The value in the RHS, namely the value bound to x (i.e. 1) is being pattern-matched against the value in the LHS, namely 1
- However, doing $2 = x$ will result in **MatchError**

Pattern Matching on More Complex Data Types

```
{name, age} = {"Bob", 25}
{_, {hour, _, _}} = :calendar.local_time()
# Mimicking Prolog's unification. Suck it Haskell!
{amount, amount, amount} = {127, 127, 127}
{amount, amount, amount} = {127, 127, 1}
```

```
[first, second, third] = [1, 2, 3]
[head | tail] = [1, 2, 3]
```

```
%{name: name, age: age} = %{name: "Bob", age: 25}
%{age: age} = %{name: "Bob", age: 25}
```

```
"Hello" <> rest = "Hello, world!"
```

Pin Operator ^

- Variables in Elixir can be rebound.
- To pattern match against an existing variable's value rather rebinding, use the pin operator ^:

```
x = 1
```

```
^x = 2
```


Where are we?

First Steps

Building Blocks

Control Flow

Pattern Matching

Conditionals

Recursion

Enumerable

Streams

Comprehensions

Pattern Matching on Function Arguments and Multi-clausal Functions

In functional languages, you can pattern match on function arguments too, and provide multiple clauses!

```
defmodule Fact do
  def fac(0), do: 1
  def fac(n), do: n * fac(n - 1)
end
```

For a function, each clause will be attempted based on the order of definition.

Function Guards

- For a factorial function, only non-negative integers are valid input.
- Using predicate functions and simple comparison expressions, we can provide constraints more than just the pattern match on each function clause:

```
defmodule Fact do
  def fac(0) do: 1
  def fac(n) when is_integer(n) and n > 0, do:
    ↪ n * fac(n - 1)
end
```

Function Guards (cont.)

- The set of operators and functions that can be called is very limited, the full list is at <https://hexdocs.pm/elixir/guards.html>
- Note that error raised in the guard expression will simply result in match failure, and Elixir will move on to the next clause, e.g. applying `length/1` on a non-list.

Multi-clausal lambdas

Anonymous functions (lambdas) may also consist of multiple clauses:

```
test_num = fn
  0 -> :zero
  x when is_number(x) and x < 0 -> :negative
  x when is_number(x) and x > 0 -> :positive
end
Enum.map(-2..2, test_num)
```

case

- **case** allows us to compare a value against many patterns until we find a matching one.
- Guards can be used also.
- If none of the clauses match, **CaseClauseError** is raised.
- Example:

```
case {1, 2, 3} do
  {4, 5, 6} -> :no_match
  {1, x, 3} when x > 0 -> :match
  _ -> :match_anything
end
x
```

case (cont.)

In fact, we can reimplement our factorial function using **case**:

```
defmodule Fact do
  def fac(n) do
    case n do
      0 ->
        1

      x when is_number(x) and x > 0 ->
        n * fac(n - 1)
    end
  end
end
```

cond

- Used to check for different conditions and find the first clause that is truthy.
- Similar to if-else if-else clauses in imperative languages.
- Note that idiomatic Elixir uses **cond** very sparingly.
- Example:

```
cond do
  2 + 2 == 5 -> :never_true
  1 + 1 == 2 -> :should_match_this
  true -> :will_match_this_if_all_fail
end
```


if and unless

- To check for only 1 condition, Elixir provides **if** and **unless** (if not). They are implemented as macros.
- If no block is specified, an implicit **nil** is returned.
- Example:

```
if nil do
  "This won't be seen"
else
  "This will"
end

unless true do
  "This will never be seen"
end
```

Where are we?

First Steps

Building Blocks

Control Flow

Pattern Matching

Conditionals

Recursion

Enumerable

Streams

Comprehensions

Our Old Friend Recursion!

```
defmodule Fact do
  def fac(0), do: 1
  def fac(n) when is_number(n) and n > 0, do: n *
    ↪ fac(n - 1)
end
```

And Tail Call Recursions!

```
defmodule Fact do
  def fac(n, acc \\ 1) when is_number(n) and
    ↪ is_number(acc) do
    case n do
      0 -> acc
      n when n > 0 -> fac(n - 1, acc * n)
    end
  end
end
```

Where are we?

First Steps

Building Blocks

Control Flow

Pattern Matching

Conditionals

Recursion

Enumerable

Streams

Comprehensions

Enumerable

- Elixir provides the concept of enumerables and the `Enum` module (<https://hexdocs.pm/elixir/Enum.html>) to work with them using higher order functions.

- Example:

```
# Allow us to use macros in Integer module
require Integer
Enum.reduce(Enum.filter(Enum.map(1..100_000,
  ↪ &(&1 * 3)), &Integer.is_even/1), 0, &+/2)
Enum.sum(Enum.filter(Enum.map(1..100_000, &(&1
  ↪ * 3)), &Integer.is_even/1))
```

The Pipe Operator

- In my opinion, the best feature that Elixir has!
- The pipe operator `|>` takes the output from the left side and passes it as the first argument to the function call on the right side.
- Similar to Unix shell `|` operator.
- Clear pipeline of transformation of data.
- Example:

```
require Integer
```

```
1..100_000
```

```
|> Enum.map(&(&1 * 3))
```

```
|> Enum.filter(&Integer.is_even/1)
```

```
|> Enum.sum()
```

Where are we?

First Steps

Building Blocks

Control Flow

Pattern Matching

Conditionals

Recursion

Enumerable

Streams

Comprehensions

Streams

- Enumerables are eager. Elixir provides a lazy alternative: the `Stream` module (<https://hexdocs.pm/elixir/Stream.html>)
- They are composable too.
- Examples:

```
stream = Stream.cycle([1, 2, 3])  
stream |> Stream.take(4) |> Enum.to_list()  
require Integer  
1..100_000 |> Stream.map(&(&1 * 3)) |>  
  ↪ Stream.filter(&Integer.is_odd/1)
```

Where are we?

First Steps

Building Blocks

Control Flow

Pattern Matching

Conditionals

Recursion

Enumerable

Streams

Comprehensions

Comprehension

- Similar concept to Python's list comprehension, but on steroids (with pattern matching)
- Example:

```
values = [good: 1, good: "a", bad: 3, good: 4]
for {good, n} when is_number(n) <- values, do:
  ↪  n * n
```
- Besides pattern matching and guards, filter expression can be used too:

```
multiple_of_3? = fn n -> rem(n, 3) == 0 end
for n <- 0..5, multiple_of_3?.(n), do: n * n
```

Comprehension (cont.)

- Comprehensions also allow multiple generators and filters:

```
dirs = ['/tmp/', '/usr/lib']  
for dir <- dirs,  
    file <- File.ls!(dir),  
    path = Path.join(dir, file),  
    File.regular?(path) do  
    File.stat!(path).size  
end
```

Comprehension (cont.)

- Using `:into`, result of a comprehension can be inserted into different data structures:

```
for <<c <- " hello world ">>, c != ?\s, into:  
  ↪  "", do: <<c>>
```

```
for {key, val} <- %{"a" => 1, "b" => 2}, into:  
  ↪  %{}, do: {key, val * val}
```

Comprehension (cont.)

Using comprehension, we can even implement quicksort!

```
defmodule Sort do
  def qsort([]) do
    []
  end

  def qsort([x | xs]) do
    qsort(for a <- xs when a < x) ++ [x] ++
    ↪ qsort(for a <- xs when a >= x)
  end
end
```

Where are we?

First Steps

Building Blocks

Control Flow

Other Useful Features

Struct

Polymorphism with Protocols

Sigil

Typespec

Struct

- Extensions built on top of maps that provide compile-time checks and default values.
- Defined using **defstruct** construct:

```
defmodule User do
  defstruct name: "John", age: 27
end

%User{}
%User{oops: :field}
jane = %User{age: 10, name: "Jane"}
%{jane | age: 11}
%{name: name} = jane
name
jane.name
is_map(jane)
```


Struct (cont.)

- In Phoenix Framework, ecto data are abstracted as a struct.
- However, ecto provides us with special constructs (using macros) to specify the struct fields, so typically in using the Phoenix Framework, we rarely need to use **defstruct**
- However, other syntaxes such as initialising a struct, updating a field, etc. are still used extensively.

Where are we?

First Steps

Building Blocks

Control Flow

Other Useful Features

Struct

Polymorphism with Protocols

Sigil

Typespec

Protocol

- A mechanism in Elixir to achieve polymorphism.
- Dispatching on a protocol is available to any data type as long as it implements the protocol.
- Defined using `defprotocol`

- Example:

```
defprotocol Size do
  @doc "Calculates the size (not the length!)
       ↪ of a data structure"
  def size(data)
end
```

- the `Size` protocol expects a 1-arity function called `size` to be implemented.

Protocol (cont.)

- Implementation is defined using `defimpl`

```
defimpl Size, for: BitString do
  def size(string), do: byte_size(string)
end

defimpl Size, for: Map do
  def size(map), do: map_size(map)
end

defimpl Size, for: Tuple do
  def size(tuple), do: tuple_size(tuple)
end
```

Protocol (cont.)

- With the protocol and implementation defined, we can start using it.
- Passing a data type that doesn't implement the protocol raises `Protocol.UndefinedError`.

```
Size.size("foo")
```

```
Size.size({:ok, "Hello"})
```

```
Size.size(%{label: "some label"})
```

```
Size.size([1, 2, 3])
```

```
Size.size(%User{})
```

- Protocols can be implemented for all Elixir data types, as well as user-defined structs:

```
defimpl Size, for: User do
```

```
  def size(_user), do: 2
```

```
end
```

Protocol (cont.)

- Elixir ships with some built-in protocols.
- For example, `Enum` module provides functions that work with any data structure that implements the `Enumerable` protocol.

Where are we?

First Steps

Building Blocks

Control Flow

Other Useful Features

Struct

Polymorphism with Protocols

Sigil

Typespec

Sigils

- Like Perl and Ruby, Elixir has sigils too.
- Strings are delimited by double-quotes.
Double-quotes inside a string must be escaped.
- These kinds of representation problems is what sigils try to solve.
- Unlike Perl and Ruby, Elixir only allows a limited set of delimiters: `//`, `||`, `"`, `'`, `()`, `[]`, `,`, `<>`

String, Charlist, Word List Sigils

- `s` sigil is used to generate string
- `c` sigil is used to generate char lists
- `w` sigil is used to generate lists of words (separated by whitespace).
- The `w` sigil also accepts the `c`, `s`, `a` modifiers (for char lists, strings and atoms respectively) to specify the data type of the elements of the resulting list.

String, Charlist, Word List Sigils (cont.)

```
~s(a string with "double quotes")  
~c|a charlist with 'single quotes' and  
  ↪ (parantheses)|  
~w(foo bar bat)  
~w(ok error)a
```

Regular Expressions

- Elixir provides Perl-compatible regexes, as implemented by the PCRE library.
- **A word of caution:** “Some poeple, when confronted with a problem, think I know, I’ll use regular expressions. Now they have two problems” (Zawinski, 1997)⁷
- There exists a **Regex** module in Elixir, as well as the regex match operator `=~`, and the `r` sigil to specify precompiled regex.

⁷There is also an interesting read at <https://blog.codinghorror.com/regular-expressions-now-you-have-two-problems/>

Regular Expressions

```
regex = ~r/foo|bar/
```

```
"foo" =~ regex
```

```
"bar" =~ regex
```

```
"bat" =~ regex
```

Where are we?

First Steps

Building Blocks

Control Flow

Other Useful Features

Struct

Polymorphism with Protocols

Sigil

Typespec

Typespec

- One advantage of using Elixir is that we can use tooling for Erlang, a 30 years old language.
- This includes type specs, a system of notating types in Erlang/Elixir.
- The Elixir compiler doesn't do type check, but one can use the **dialyzer** tool to perform type check.
- Owing to its age, **dialyzer** is more mature than, say, **mypy**, especially in terms of type inference⁸

⁸More information on Erlang type inference in https://it.uu.se/research/group/hipe/papers/succ_types.pdf

Typespec (cont.)

- Not compulsory, but being able to read type spec helps a lot in reading documentation.
- Some code that I wrote in Source Academy have type specs.
- In source code, typically notated using module attribute `@spec`
- More information in <https://hexdocs.pm/elixir/typespecs.html>

Where are we?

First Steps

Building Blocks

Control Flow

Other Useful Features

Source Academy Backend: Cadet

Structure

Structure

- Phoenix Framework:
 - Router
 - Plugs
- MVC
 - Model
 - Contexts: inside `lib/cadet/`
 - Migrations: inside `priv/repo/migrations/`
 - Seeds: inside `priv/repo/seeds.exs`
 - View: `lib/cadet_web/views/` – renders json
 - Controller: inside `lib/cadet_web/controllers/`

Structure (Cont.)

- Jobs: inside `lib/cadet/jobs/`
 - Updater, XML Parser
 - Autograder
- Mix tasks for convenience: inside `lib/mix/tasks/`
- Config files: inside `config/`
- Tests: 98% test coverage. Every feature are tested.