Growing with Elm: An Introduction to Functional Programming

Alex Simonian 5-18-2017

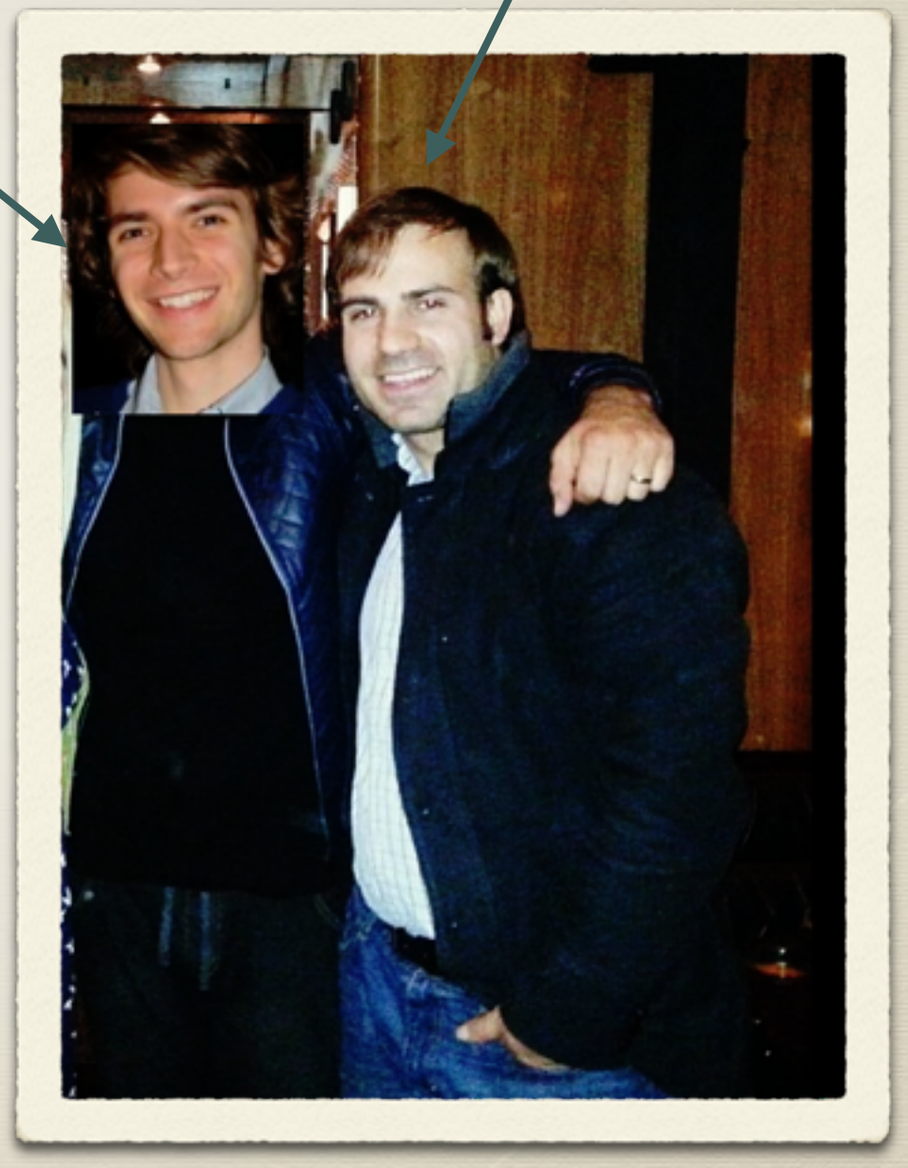
elm

Special thanks to Anderson Cook

What is Elm?

Me!

Evan!



Created by Evan Czaplicki, Elm is a concurrent functional programming language built from the ground up to be the best programming language for building UIs.

How did we get here?

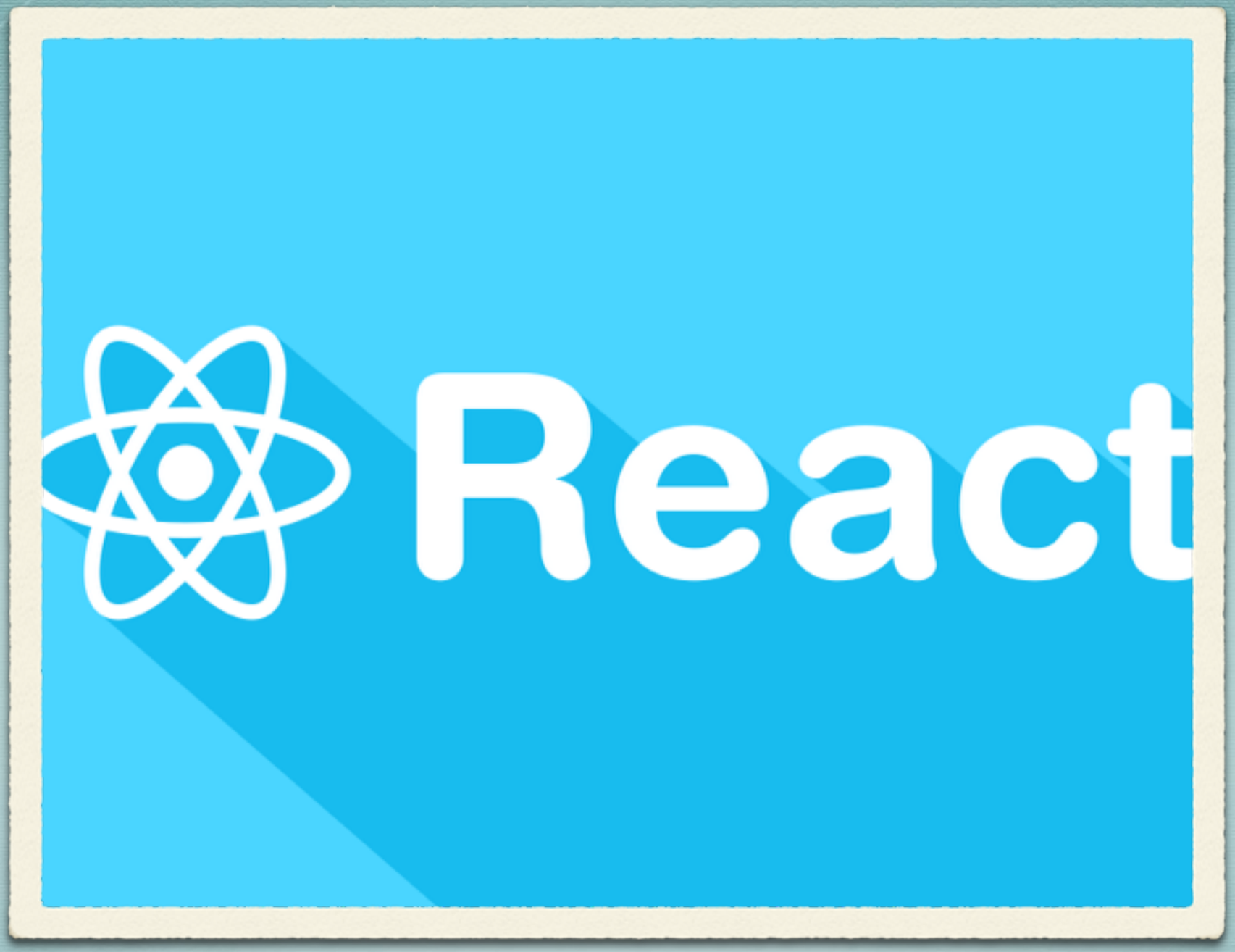
DOM manipulation, much easier now.

What failed and why we did we move on

* State was stored in the DOM
* To get state, we'd re-query that DOM
* Querying the DOM is slow
* State had to be stored in the DOM
* Or we'd call an API to get brand new DOM
* Where is the state?

And then…

2-way data-binding, template library,
state in DOM and code



React's built-in state management

State no longer built into the DOM

What failed and why did we move on

* Somewhat controlled, but ultimately imperative and mutative API
* Performance cost of not using pure render
* Like jQuery, the data is tied to the element
* Component ‘owning’ data isn’t cool
* Passing around state/props — yikes!

Redux wins

* Encourages you not to use local state
* Ignoring most parts of React and using React just to render stateless components
* Single state atom (the store)
* Query stores, never component state
* React becomes a reflection of the state
* Where did Redux come from?

Elm to the rescue

* Purely functional language for front-end development
* Derived from ML
* Statically-typed
* Expression-based
* Compiles to JS
* No run-time errors
* Friendly compiler errors
* Declarative

Development Toolkit

JavaScript	Elm Platform
Babel	Elm (compiler)
Redux	Elm (design pattern)
Flow	Elm (type annotation)
Immutable, Ramda, Lodash	Elm (stdlib)
React	html
ESLint	elm-format
Mocha, Chai	elm-test

Elm vs JavaScript Comparison

Impure and Mutable

```
3
4
5
6   let name = 'Alex';
7
8   function getName() {
9       return name;
10  }
11
12  function setName(newName) {
13      name = newName;
14  }
```

Built on Functions

Modular and Reusable

Functional

```
> greet name = "Hello, " ++ name
<function> : String -> String
```

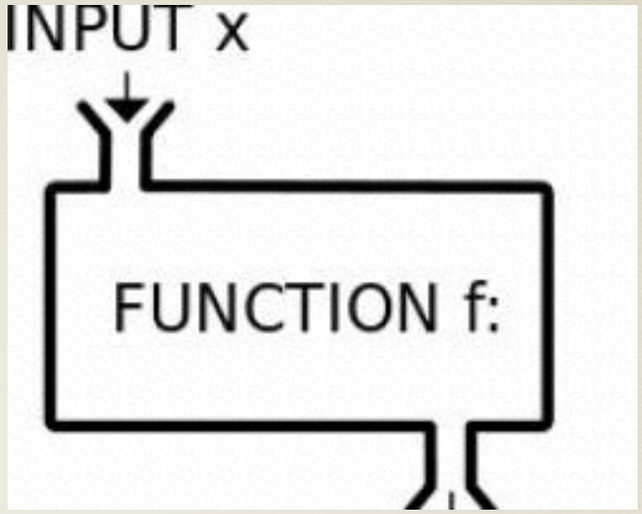
```
> greet "Nashville Beginners Meetup"
"Hello, Nashville Beginners Meetup" : String
```

```
> add x y = x + y
<function> : number -> number -> number
```

```
> add 2 2
4 : number
```

Pure and Immutable

```
> add 2 2
4 : number
```




```
>>> names = ['Alex', 'Bill', 'Cat']
>>> names[0] = 'Aric'
>>> names
['Aric', 'Bill', 'Cat']
```

Python (mutable)

```
> names = ["Alex", "Bill", "Cat"]
["Alex","Bill","Cat"] : List String
> names[0] = "Aric"
================================ ERRORS

-- PARTIAL PATTERN ------------------------
```

Elm (immutable)

Declarative vs Imperative

* **Declarative** be like : tell me WHAT to do. Write our SQL code, let the engine handle the implementation details.
* **Imperative** be like : tell me HOW to do it. Procedural, Object-oriented. Mutating state, impure functions.

Imperative

```
function doubleNumbers(numbers) {
    const doubled = [];
    const l = numbers.length;

    for (let i = 0; i < l; i++) {
        doubled.push(numbers[i] * 2);
    }

    return doubled;
}

doubleNumbers(([1, 2, 3, 4, 5]));
// [2, 4, 6, 8, 10]
```

Declarative

Linked Lists cannot index into

```
[1, 2, 3, 4, 5].map( num => num * 2 )

[2, 4, 6, 8, 10]
```

```
> myList = [1, 2, 3, 4, 5]
[1,2,3,4,5] : List number
>
> double n = n * 2
<function> : number -> number
>
> doubleNumbers list = List.map double list
<function> : List number -> List number
>
> doubleNumbers myList
[2,4,6,8,10] : List number
```

Currying on building blocks: partially applied functions

```
> add x y z = x + y + z
<function> : number -> number -> number -> number
```

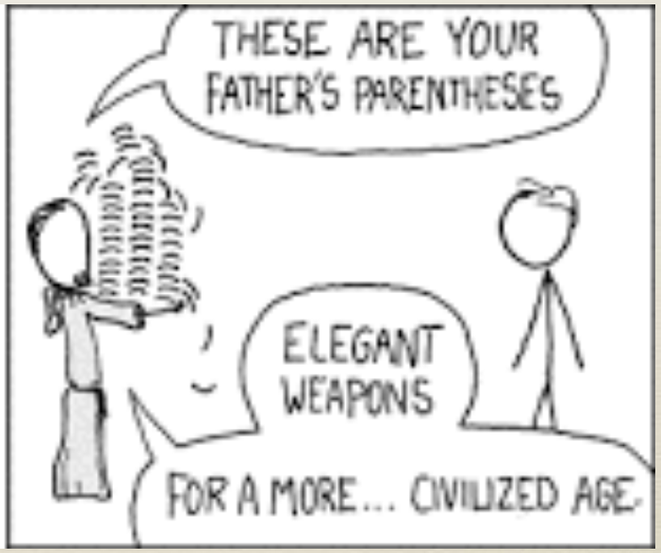
```
> add 1 2 3
6 : number
```

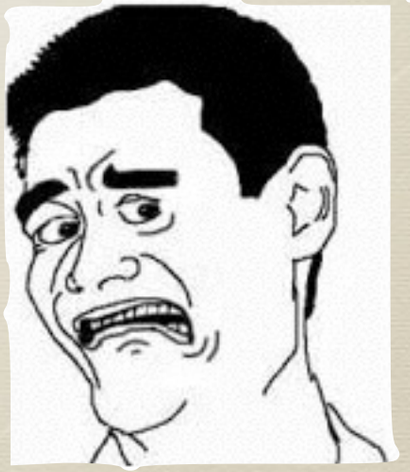
```
> add1 = add 1
<function> : number -> number -> number
> add3 = add1 2
<function> : number -> number
> add3 3
6 : number
```

Piping

```
list = List.range 1 10
square n = n * n


List.map square (List.filter ((<) 6) (List.map ((*) 2) list))
```





Piping

```
list = List.range 1 10
square n = n * n


List.map square (List.filter ((<) 6) (List.map ((*) 2) list))
```

```
list = List.range 1 10
square n = n * n


list
    |> List.map ((*) 2)
    |> List.filter ((<) 6)
    |> List.map square
```

Evaluate from
INSIDE to OUTSIDE

Static and Strong Typing

If the types do not match, then it doesn't compile.

Static and Strong Typing

JavaScript be like:

```
> '1' + 1
<· "11"
```

Elm be like:

```
> '1' + 1
-- TYPE MISMATCH -----------------------------------

The left argument of (+) is causing a type mismatch.

3|   '1' + 1
     ^^^
(+) is expecting the left argument to be a:

    number

But the left argument is:

    Char
```

Type annotations are contractual

```
sayName : String -> String
sayName name =
    "Hi, " ++ name

divide2 : Float -> Float -> Float
divide2 x y =
    x / y
```

Tuples

```
car : ( String, Int )
car =
    ( "Acura", 2002 )


make =
    Tuple.first car      -- "Acura"


year =
    Tuple.second car     -- 2002
```

Records

```
car : { make : String, model : String, year : Int }
car =
    { make = "Acura"
    , model = "RSX"
    , year = 2002
    }

car.make          -- "Acura"
car.model         -- "RSX"
car.year          -- 2002
.make car         -- "Acura"
```

Type alias

```
type alias Car =
    { make : String
    , model : String
    , year : Int
    }


car : Car
car =
    Car "Acura" "RSX" 2002
```

Immutable state

```
car : Car
car =
    Car "Acura" "RSX" 2002


olderCar = { car | year = car.year + 1 }

car.year            -- 2002
olderCar.year       -- 2003
```

Similar to our JavaScript buddies

Spread syntax

```
options = {...optionsDefault, ...options};
```

Object.assign()

```
options = Object.assign({}, optionsDefault, options);
```

Union Types

```
type alias Car =
    { make : String
    , model : String
    , year : Int
    }


car : Car
car =
    Car "Acura" "RSX" 2002
```

```
type Model
    = RSX
    | TSX
    | MDX
```

```
car1 = Car "Acura" RSX 2002
car2 = Car "Acura" TSX 2008
car3 = Car "Acura" MDX 2010
```

Comparable to Enums

How Elm deals with null

```
type Maybe a
    = Just a
    | Nothing
```

How Elm deals with null

```
divide : number -> number -> Maybe Float
divide x y =
    if y == 0 then
        Nothing
    else
        Just (x / y)


divide 4 2        -- Just 2
divide 4 0        -- Nothing
```

How Elm deals with null

```
case divide 4 2 of
    Just n ->
        "Result is " ++ (toString n)

    Nothing ->
        "No Result"
```

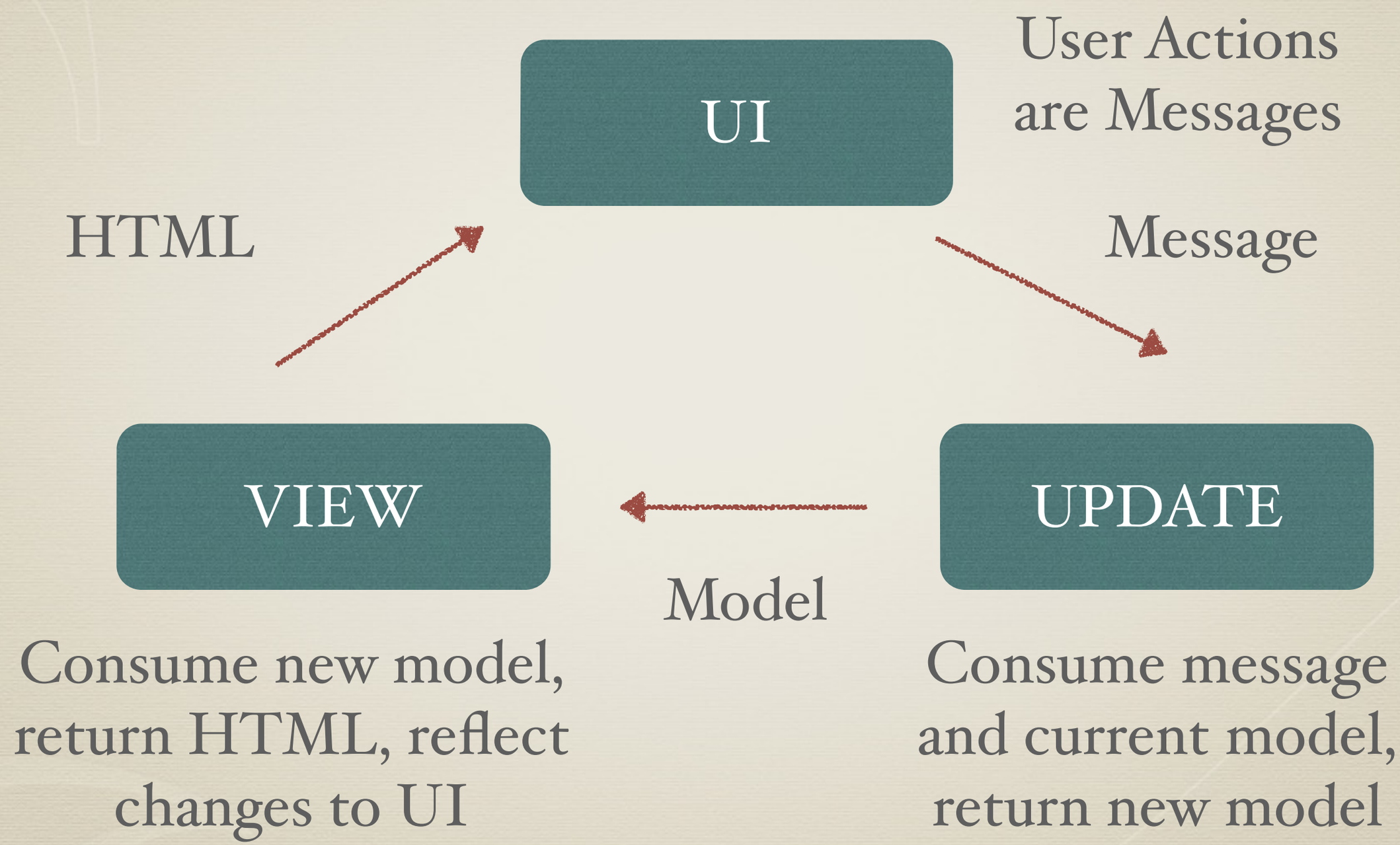
Virtual DOM

No template library!

http://mbylstra.github.io/html-to-elm/

attributes child nodes

Elm Architecture



Elm Starter Pack

https://github.com/asimonia/Elm-Starter

- Package Manager
- Compiler
- REPL
- Dev Server

Elm Resources

* Front End Masters: https://frontendmasters.com/
* Elm Examples: http://elm-lang.org/examples
* Pragmatic Studio: https://pragmaticstudio.com/

CONTACT ME

Email: alex.simonian@gmail.com
Portfolio: www.alexsimonian.com
LinkedIn: linkedin.com/in/alex-simonian