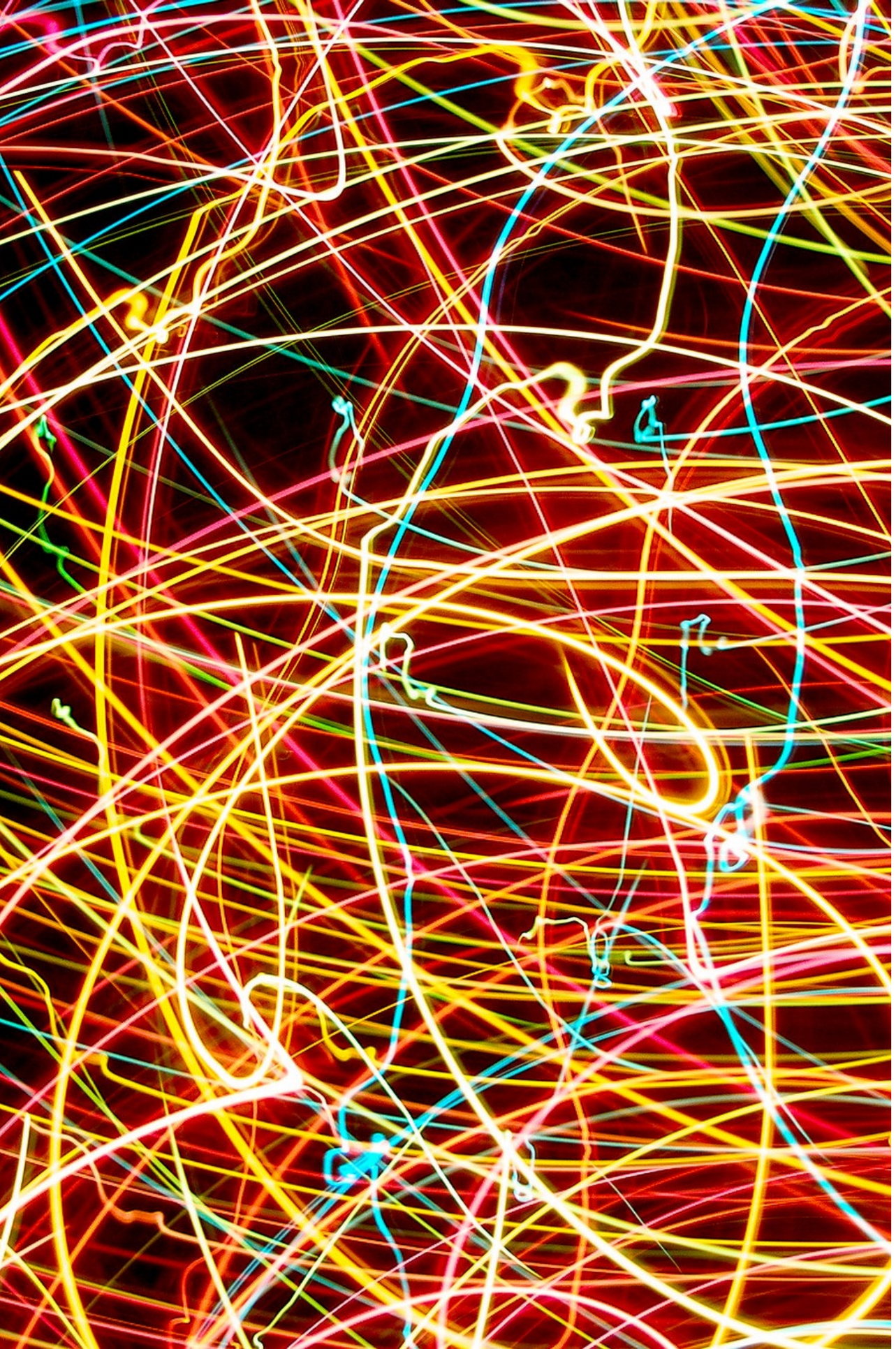
STEPPING OUT OF THE CHAOS

With Elm

STEPPING OUT OF THE CHAOS

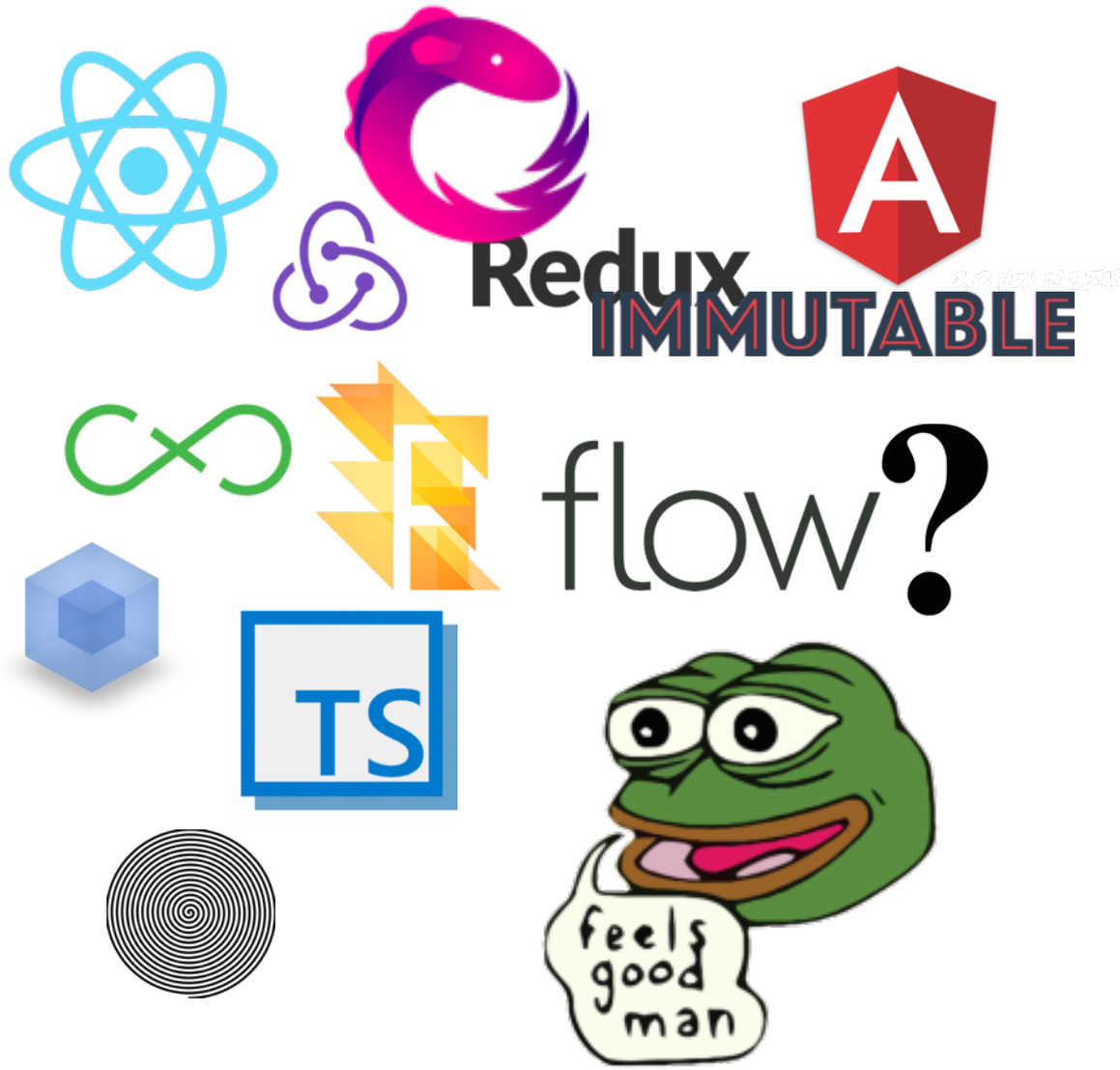
With Elm

THE CHAOTIC JS WORLD



TOO MANY LIBRARIES!

- When starting a project it is not clear what combination of libraries should be picked.



WHAT IS JAVASCRIPT'S SINGLE BIGGEST ISSUE?

MAYBE ITS QUIRKINESS?

```
> function dis() { return this }
undefined
> five = dis.call(5)
[Number: 5]
> five.wtf = 'potato'
'potato'
> five.wtf
'potato'
> five * 5
25
> five.wtf
'potato'
> five++
5
> five.wtf
undefined
> five.wtf = 'potato?'
'potato?'
> five.wtf
undefined
> five
6
>
```

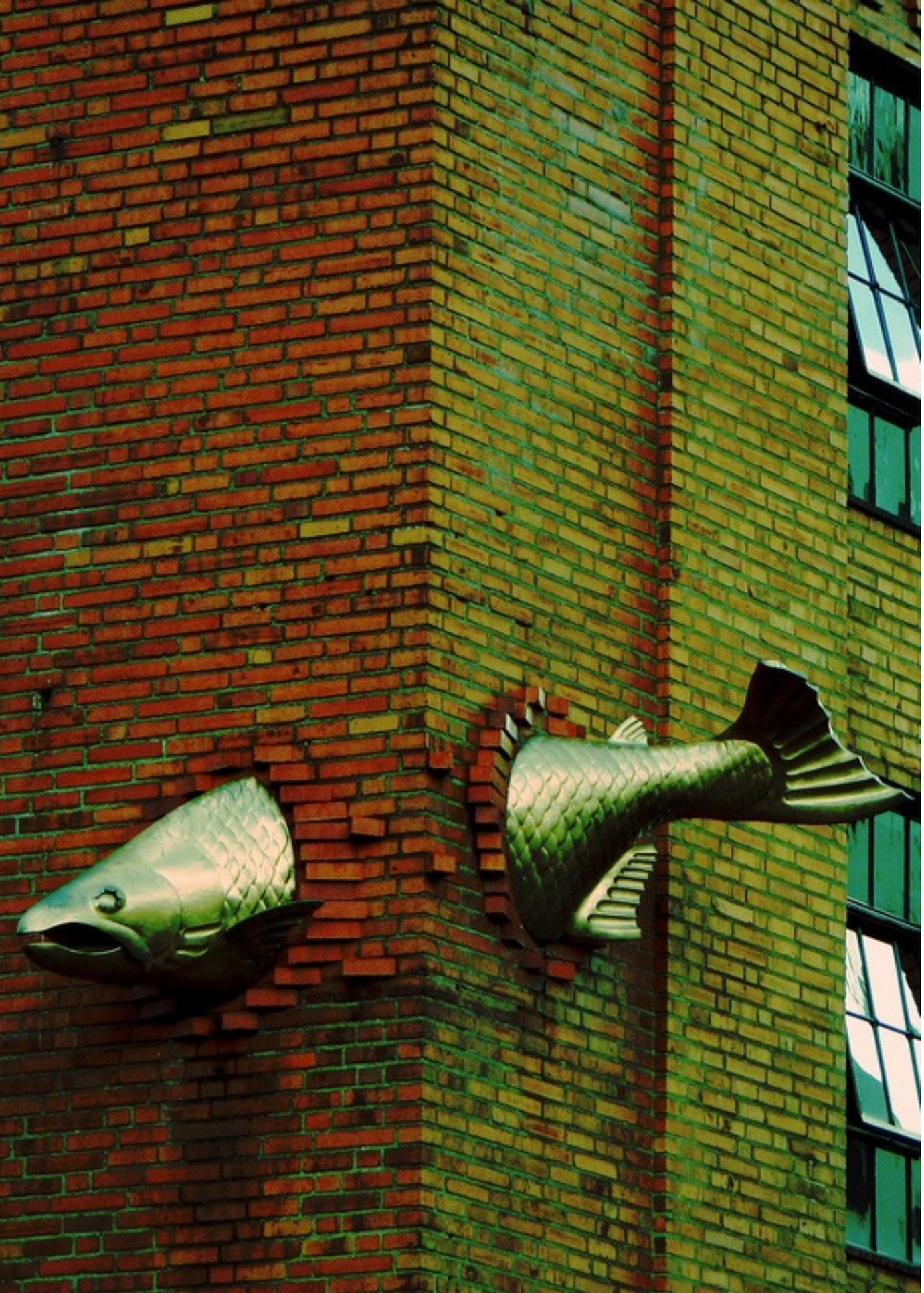
MAYBE ITS QUIRKINESS?

…Not really

```
> function dis() { return this }
undefined
> five = dis.call(5)
[Number: 5]
> five.wtf = 'potato'
'potato'
> five.wtf
'potato'
> five * 5
25
> five.wtf
'potato'
> five++
5
> five.wtf
undefined
> five.wtf = 'potato?'
'potato?'
> five.wtf
undefined
> five
6
>
```

IT IS:
MAINTAINABILITY

IT IS:
MAINTAINABILITY



JS IS EASY TO GET STARTED WITH

- But it is also difficult to **maintain** in the long run.

JS IS EASY TO GET STARTED WITH

```
> var x = document.getElementById('form')
✕ ▶ Uncaught TypeError: document.getElementById is
  not a function(…)
```

- But it is also difficult to **maintain** in the long run.
- A wide array of runtime errors can happen due to simple **overlooks**

JS IS EASY TO GET STARTED WITH

```
> var x = document.getElementById('form')
✕ ▶ Uncaught TypeError: document.getElementById is
  not a function(…)
```

- But it is also difficult to **maintain** in the long run.
- A wide array of runtime errors can happen due to simple **overlooks**

This should never happen

We've found 994,196 code results

MaziMuhlari/testimonyapp – multiple.js
Showing the top five matches. Last indexed on Mar 31.

```
1   import $ from 'jquery';
2
3   var ParsleyMultiple = function () {
4     this.__class__ = 'ParsleyFieldMultiple';
…
70      if (this.$element.is('select') && null === this.$element.val())
71        return [];
72
73      // Default case that should never happen
74      return this.$element.val();
```

Hepolise/Chromium-Snapdragon-caf – canvas-save-restore-with-path.js
Showing the top four matches. Last indexed on Mar 31.

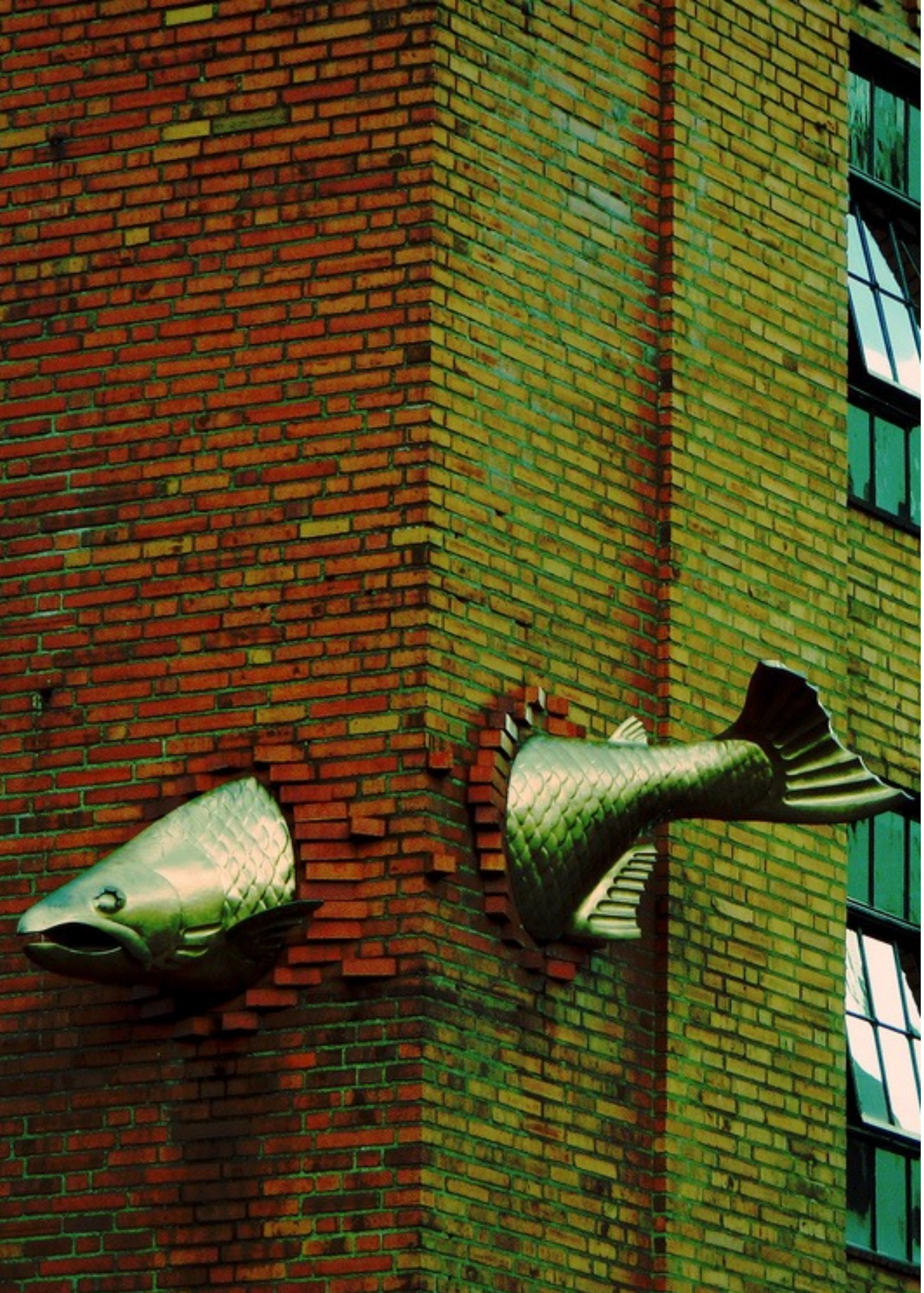
```
1   description("This test ensures that paths are correctly handled over sa
…
11      var data = context.getImageData(x,y,1,1);
12      if (!data) // getImageData failed, which should never happen
```

corvidian/tenttiarkisto – multiple.js
Showing the top five matches. Last indexed on Mar 25.

```
68        if (this.$element.is('select') && null === this.$element.val())
69          return [];
70
```

JS IS EASY TO GET STARTED WITH

- But it is also difficult to **maintain** in the long run.
- A wide array of runtime errors can happen due to simple **overlooks**
- It is easier to come up with hacks for solving issues. Hacks are **rarely** documented.



JS IS EASY TO GET STARTED WITH

- But it is also difficult to **maintain** in the long run.
- A wide array of runtime errors can happen due to simple **overlooks**
- It is easier to come up with hacks for solving issues. Hacks are **rarely** documented.
- TypeScript, flow, Rx, unit tests… they all help, **but** they are only used by people that **understand** their value.

EXPERIENCE AND DISCIPLINE

The magic combo to make javascript maintainable

```
function reducer(comments, action) {
  switch (action) {
      case 'ADD_LIKE':
        return {
          ...comments,
          likes: comments.likes + 1
        }
    }
```

EXPERIENCE AND DISCIPLINE

```
1 function reducer(comments, action) {
2   switch (action) {
3       case 'ADD_LIKE':
4         comments.likes++;
5         return state;
6   }
7 }
8
```

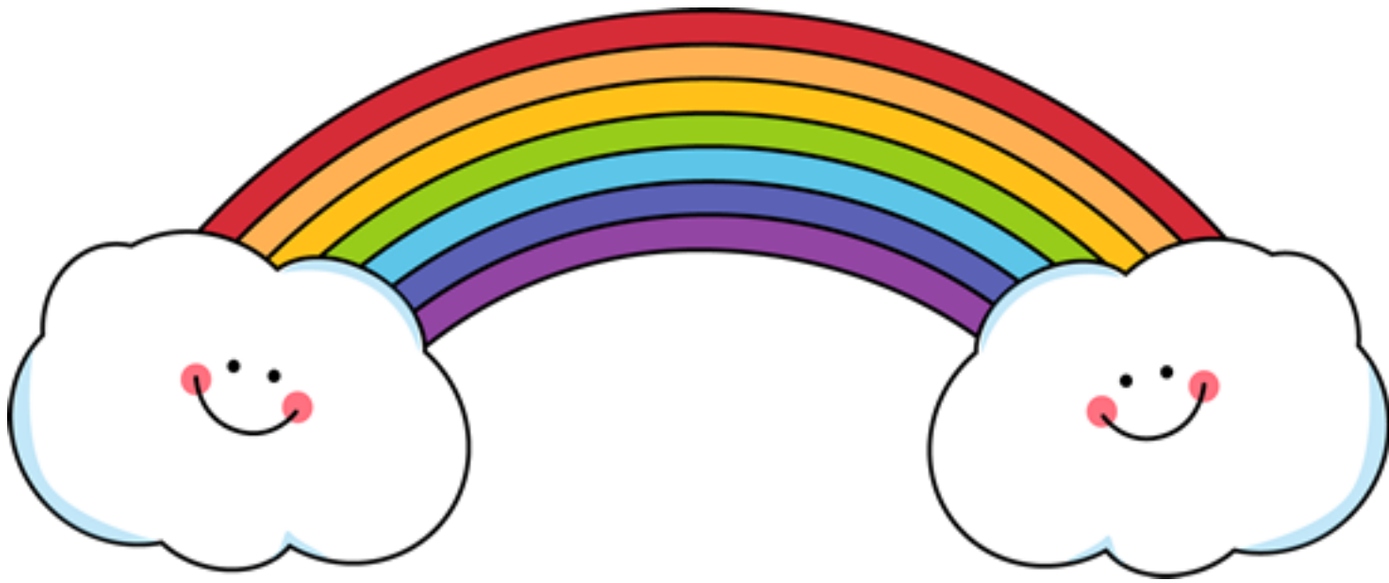
VS TIGHT DEADLINE

STEPPING OUT OF THE CHAOS

With Elm

STEPPING OUT OF THE CHAOS

With Elm



> What I want to do is: I want to get to this magic realm where we get that [insanely high] level of **maintainability**, but where we **also** have something **usable**.
>
> *-Evan Czaplicki, creator of **Elm***

CAN YOU IMAGINE NEVER HAVING A RUNTIME ERROR?

CAN YOU IMAGINE NEVER HAVING A RUNTIME ERROR?

```elm
import List


someNumbers =
  [ 1, 2, 3, 4 ]


isEven num =
  num % 2 == 0


oddNumbers =
  List.filtre isEven someNumbers

```

Module importing

Defining a Value

Defining a Function

Calling a function with a value

A simple Elm program

```elm
import List


someNumbers =
  [ 1, 2, 3, 4 ]


isEven num =
  num % 2 == 0


oddNumbers =
  List.filtre isEven someNumbers

```

Module importing

Defining a Value

Defining a Function

Calling a function with a value

A simple Elm program

```
Detected errors in 1 module.


-- NAMING ERROR ----------------------------------------------------

Cannot find variable `List.filtre`.

15|    List.filtre isEven someNumbers
       ^^^^^^^^^^^^

`List` does not expose `filtre`. Maybe you want one of the following?

    List.filter
```

```
Detected errors in 1 module.


-- NAMING ERROR ----------------------------------------------------

Cannot find variable `List.filtre`.

15|    List.filtre isEven someNumbers
       ^^^^^^^^^^^^

`List` does not expose `filtre`. Maybe you want one of the following?

    List.filter
```

```elm
type alias Person =
  { name : String
  , phoneNumber : String
  , address : String
  , age : Int
  }


changeAddress : Person -> String -> Person
changeAddress person newAddress =
  { person | adress = newAddress }
```

```elm
type alias Person =
  { name : String
  , phoneNumber : String
  , address : String
  , age : Int
  }
```

Declare a new data type

```elm
changeAddress : Person -> String -> Person
changeAddress person newAddress =
  { person | adress = newAddress }
```

```elm
type alias Person =
  { name : String
  , phoneNumber : String
  , address : String
  , age : Int
  }
```

Declare a new data type

Properties of the Person

```elm
changeAddress : Person -> String -> Person
changeAddress person newAddress =
  { person | adress = newAddress }
```

```elm
type alias Person =
  { name : String
  , phoneNumber : String
  , address : String
  , age : Int
  }
```

Declare a new data type

Properties of the Person

```elm
changeAddress : Person -> String -> Person
changeAddress person newAddress =
  { person | adress = newAddress }
```

Type annotation

```elm
type alias Person =
  { name : String
  , phoneNumber : String
  , address : String
  , age : Int
  }
```

Declare a new data type

Properties of the Person

```elm
changeAddress : Person -> String -> Person
changeAddress person newAddress =
  { person | adress = newAddress }
```

Type annotation

Takes the person argument and sets the property to newAddress

```elm
type alias Person =
  { name : String
  , phoneNumber : String
  , address : String
  , age : Int
  }
```

Declare a new data type

Properties of the Person

```elm
changeAddress : Person -> String -> Person
changeAddress person newAddress =
  { person | adress = newAddress }
```

Type annotation

Takes the person argument and sets the property to newAddress

This function actually returns a different person.

It does not change the passed argument!

```
-- TYPE MISMATCH ---------------------------------------------------

The type annotation for `changeAddress` does not match its definition.

14| changeAddress : Person -> String -> Person
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^

The type annotation is saying:

    { address : ..., age : ..., name : ..., phoneNumber : ... }
    -> String
    -> { address : ..., age : ..., name : ..., phoneNumber : ... }

But I am inferring that the definition has this type:

    { b | adress : ... } -> c -> { b | adress : ... }

Hint: I compared the record fields and found some potential typos.

    address <-> adress
```

```
-- TYPE MISMATCH ---------------------------------------------------

The type annotation for `changeAddress` does not match its definition.

14| changeAddress : Person -> String -> Person
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^

The type annotation is saying:

    { address : ..., age : ..., name : ..., phoneNumber : ... }
    -> String
    -> { address : ..., age : ..., name : ..., phoneNumber : ... }

But I am inferring that the definition has this type:

    { b | adress : ... } -> c -> { b | adress : ... }

Hint: I compared the record fields and found some potential typos.

    address <-> adress
```

Tak for hjælpen

YOU DON'T NEED TO TELL HOW THE DOM IS CHANGED

just declare how it should look like

```elm
import Html exposing (..)
import Html.Attributes exposing (..)
import List


languages =
  [ "Javascript", "Elixir", "Elm" ]


languageItem language =
  li [ class "lang" ] [ text language ]


main =
  ul
    [ class "languages" ]
    (List.map languageItem languages)


```

```elm
import Html exposing (..)
import Html.Attributes exposing (..)
import List


languages =
  [ "Javascript", "Elixir", "Elm" ]


languageItem language =
  li [ class "lang" ] [ text language ]


main =
  ul
    [ class "languages" ]
    (List.map languageItem languages)
```

*This builds a single *

```elm
import Html exposing (..)
import Html.Attributes exposing (..)
import List


languages =
  [ "Javascript", "Elixir", "Elm" ]


languageItem language =
  li [ class "lang" ] [ text language ]


main =
  ul
    [ class "languages" ]
    (List.map languageItem languages)


```

*This builds a single *

Html is constructed declaratively

```elm
import Html exposing (..)
import Html.Attributes exposing (..)
import List


languages =
  [ "Javascript", "Elixir", "Elm" ]


languageItem language =
  li [ class "lang" ] [ text language ]


main =
  ul
    [ class "languages" ]
    (List.map languageItem languages)


```

*This builds a single *

Html is constructed declaratively

First argument is a list of attributes

```elm
import Html exposing (..)
import Html.Attributes exposing (..)
import List


languages =
  [ "Javascript", "Elixir", "Elm" ]


languageItem language =
  li [ class "lang" ] [ text language ]


main =
  ul
    [ class "languages" ]
    (List.map languageItem languages)

```

*This builds a single *

Html is constructed declaratively

First argument is a list of attributes

Second argument is a list of children Html Nodes

```elm
import Html exposing (..)
import Html.Attributes exposing (..)
import List


languages =
  [ "Javascript", "Elixir", "Elm" ]


languageItem language =
  li [ class "lang" ] [ text language ]


main =
  ul
    [ class "languages" ]
    (List.map languageItem languages)
```

```html
▼<ul class="languages">
    <li class="lang">Javascript</li>
    <li class="lang">Elixir</li>
    <li class="lang">Elm</li>
  </ul>
```

*This builds a single *

Html is constructed declaratively

First argument is a list of attributes

Second argument is a list of children Html Nodes

YOU DON'T NEED TO HANDLE EVENTS

just declare the message they represent

```elm
import Html exposing (..)
import Html.App exposing (beginnerProgram)
import Html.Events exposing (..)
import List


main =
  beginnerProgram { model = [], view = view, update = update }


type Msg
  = Add
  | Remove


view langs =
  div
    []
    [ ul [] (List.map languageItem langs)
    , button [ onClick Add ] [ text "Add" ]
    , button [ onClick Remove ] [ text "Remove" ]
    ]

```

```elm
import Html exposing (..)
import Html.App exposing (beginnerProgram)
import Html.Events exposing (..)
import List


main =
  beginnerProgram { model = [], view = view, update = update }


type Msg
  = Add
  | Remove


view langs =
  div
    []
    [ ul [] (List.map languageItem langs)
    , button [ onClick Add ] [ text "Add" ]
    , button [ onClick Remove ] [ text "Remove" ]
    ]

```

Does the initial wiring for you

```elm
import Html exposing (..)
import Html.App exposing (beginnerProgram)
import Html.Events exposing (..)
import List


main =
  beginnerProgram { model = [], view = view, update = update }


type Msg
  = Add
  | Remove


view langs =
  div
    []
    [ ul [] (List.map languageItem langs)
    , button [ onClick Add ] [ text "Add" ]
    , button [ onClick Remove ] [ text "Remove" ]
    ]

```

Does the initial wiring for you

we start the app with an empty list

```elm
import Html exposing (..)
import Html.App exposing (beginnerProgram)
import Html.Events exposing (..)
import List


main =
  beginnerProgram { model = [], view = view, update = update }


type Msg
  = Add
  | Remove


view langs =
  div
    []
    [ ul [] (List.map languageItem langs)
    , button [ onClick Add ] [ text "Add" ]
    , button [ onClick Remove ] [ text "Remove" ]
    ]

```

Does the initial wiring for you

we start the app with an empty list

The view function in this file

```elm
import Html exposing (..)
import Html.App exposing (beginnerProgram)
import Html.Events exposing (..)
import List


main =
  beginnerProgram { model = [], view = view, update = update }


type Msg
  = Add
  | Remove


view langs =
  div
    []
    [ ul [] (List.map languageItem langs)
    , button [ onClick Add ] [ text "Add" ]
    , button [ onClick Remove ] [ text "Remove" ]
    ]

```

Does the initial wiring for you

we start the app with an empty list

The view function in this file

Defines the type of messages the app will get from the user

```elm
import Html exposing (..)
import Html.App exposing (beginnerProgram)
import Html.Events exposing (..)
import List


main =
  beginnerProgram { model = [], view = view, update = update }


type Msg
  = Add
  | Remove


view langs =
  div
    []
    [ ul [] (List.map languageItem langs)
    , button [ onClick Add ] [ text "Add" ]
    , button [ onClick Remove ] [ text "Remove" ]
    ]

```

Does the initial wiring for you

we start the app with an empty list

The view function in this file

Defines the type of messages the app will get from the user

Pass the message on click

```elm
import Html exposing (..)
import Html.App exposing (beginnerProgram)
import Html.Events exposing (..)
import List


main =
  beginnerProgram { model = [], view = view, update = update }


type Msg
  = Add
  | Remove


view langs =
  div
    []
    [ ul [] (List.map languageItem langs)
    , button [ onClick Add ] [ text "Add" ]
    , button [ onClick Remove ] [ text "Remove" ]
    ]

```

Does the initial wiring for you

we start the app with an empty list

The view function in this file

Defines the type of messages the app will get from the user

Pass the message on click

```elm
languages =
  [ "Javascript", "Elixir", "Elm", "PHP", "Ruby" ]


update msg langs =
  let
    length =
      List.length langs
  in
    case msg of
      Add ->
        languages
          |> List.drop length
          |> List.take 1
          |> List.append langs

      Remove ->
        langs
          |> List.take (length - 1)


```

```
27 languages =
28   [ "Javascript", "Elixir", "Elm", "PHP", "Ruby" ]
29
30
31 update msg langs =
32   let
33     length =
34       List.length langs
35   in
36     case msg of
37       Add ->
38         languages
39           |> List.drop length
40           |> List.take 1
41           |> List.append langs
42
43       Remove ->
44         langs
45           |> List.take (length - 1)
46
47
48
```

The message from the user

```elm
languages =
  [ "Javascript", "Elixir", "Elm", "PHP", "Ruby" ]


update msg langs =
  let
    length =
      List.length langs
  in
    case msg of
      Add ->
        languages
          |> List.drop length
          |> List.take 1
          |> List.append langs

    Remove ->
      langs
        |> List.take (length - 1)
```

The message from the user

The "model" (list of languages)

```elm
languages =
  [ "Javascript", "Elixir", "Elm", "PHP", "Ruby" ]


update msg langs =
  let
    length =
      List.length langs
  in
    case msg of
      Add ->
        languages
          |> List.drop length
          |> List.take 1
          |> List.append langs

      Remove ->
        langs
          |> List.take (length - 1)
```

The message from the user

The "model" (list of languages)

scopes variables to this function

```elm
languages =
  [ "Javascript", "Elixir", "Elm", "PHP", "Ruby" ]


update msg langs =
  let
    length =
      List.length langs
  in
    case msg of
      Add ->
        languages
          |> List.drop length
          |> List.take 1
          |> List.append langs

      Remove ->
        langs
          |> List.take (length - 1)
```

The message from the user

The "model" (list of languages)

scopes variables to this function

These are always safe operations. Even on empty lists

```elm
languages =
  [ "Javascript", "Elixir", "Elm", "PHP", "Ruby" ]


update msg langs =
  let
    length =
      List.length langs
  in
    case msg of
      Add ->
        languages
          |> List.drop length
          |> List.take 1
          |> List.append langs

      Remove ->
        langs
          |> List.take (length - 1)
```

The message from the user

The "model" (list of languages)

scopes variables to this function

These are always safe operations. Even on empty lists

Finally return the modified "model"

```elm
languages =
  [ "Javascript", "Elixir", "Elm", "PHP", "Ruby" ]


update msg langs =
  let
    length =
      List.length langs
  in
    case msg of
      Add ->
        languages
          |> List.drop length
          |> List.take 1
          |> List.append langs

      Remove ->
        langs
          |> List.take (length - 1)
```

The message from the user

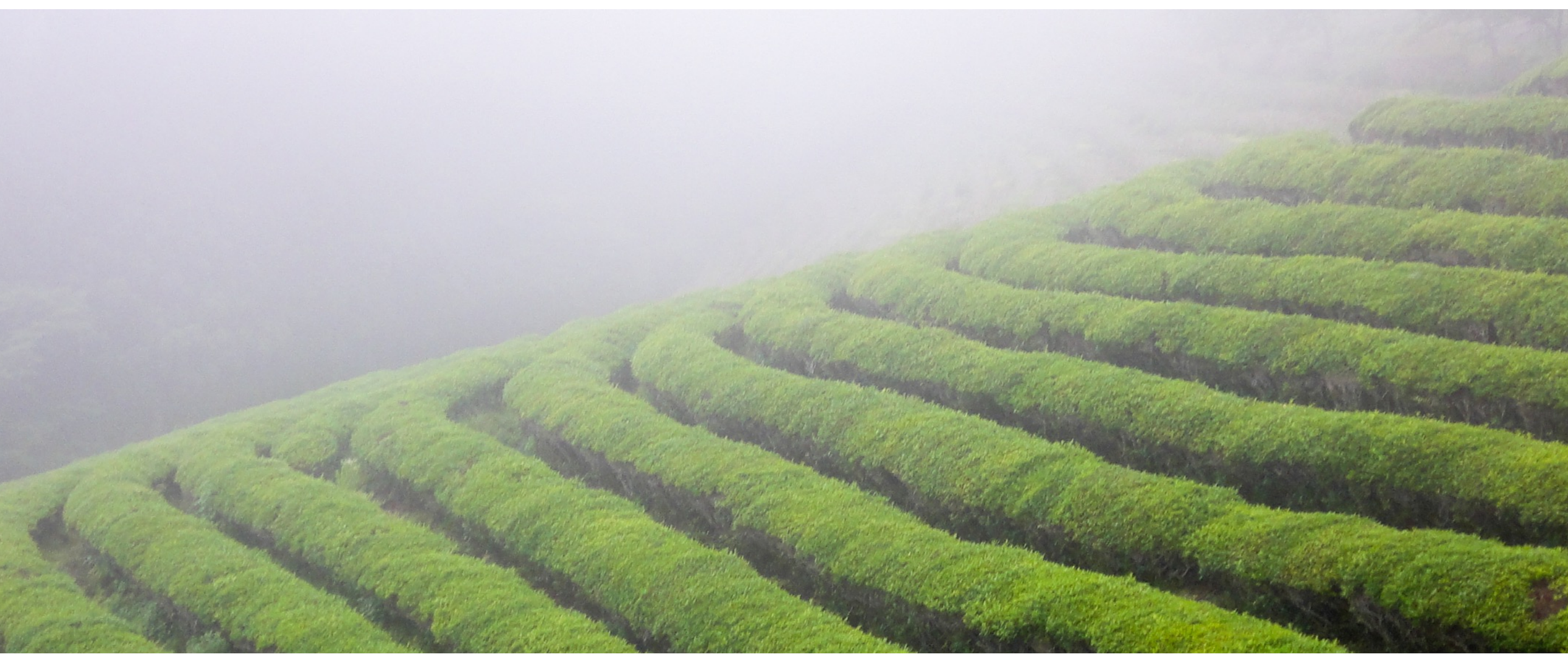
The "model" (list of languages)

scopes variables to this function

These are always safe operations. Even on empty lists

Finally return the modified "model"

The pipe is like the "." operator in JS



WOULD YOU LIKE SOME TEA?

TEA: The Elm Architecture

```
main =
  beginnerProgram { model = [], view = view, update = update }
```

DO YOU REMEMBER THIS?

```
main =
  beginnerProgram { model = [], view = view, update = update }
```

Idea: It would be nice if we could have composable and reusable modules by only having to declare these 3 things.

DO YOU REMEMBER THIS?

```elm
module Langs exposing (Model, Msg, view, update)¬

import Html exposing (..)
import Html.Events exposing (..)
import List


type alias Model =
  { items : List String, db : List String }


update : Msg -> Model -> Model
update msg model =
  let
    length =
      List.length model.items
  in
    case msg of
      Add ->
        { model | items = appendLang length model }

      Remove ->
        { model | items = removeLang length model }
```

Declare a module with the TEA API

Instead of hardcoding the languages. Make them dynamic

Refactored the previous code to use helper functions

```elm
module Main exposing (..)¬

import Langs
import Html exposing (..)
import Html.App exposing (beginnerProgram, map)


init =
  { frontend = { items = [], db = [ "Js", "Elm", "CofeeScript", "CSS" ] }
  , backend = { items = [], db = [ "Ruby", "Java", "PHP", ".Net" ] }
  }


type alias Model =
  { frontend : Langs.Model, backend : Langs.Model }


type Msg
  = Frontend Langs.Msg
  | Backend Langs.Msg


update msg model =
  case msg of
    Frontend frontendMsg ->
      { model | frontend = Langs.update frontendMsg model.frontend }

    Backend backendMsg ->
      { model | backend = Langs.update backendMsg model.backend }

```

```elm
module Main exposing (..)¬

import Langs
import Html exposing (..)
import Html.App exposing (beginnerProgram, map)


init =
  { frontend = { items = [], db = [ "Js", "Elm", "CofeeScript", "CSS" ] }
  , backend = { items = [], db = [ "Ruby", "Java", "PHP", ".Net" ] }
  }


type alias Model =
  { frontend : Langs.Model, backend : Langs.Model }


type Msg
  = Frontend Langs.Msg
  | Backend Langs.Msg


update msg model =
  case msg of
    Frontend frontendMsg ->
      { model | frontend = Langs.update frontendMsg model.frontend }

    Backend backendMsg ->
      { model | backend = Langs.update backendMsg model.backend }

```

We'll have two list of languages. Here are the defaults

```elm
module Main exposing (..)¬

import Langs
import Html exposing (..)
import Html.App exposing (beginnerProgram, map)


init =
  { frontend = { items = [], db = [ "Js", "Elm", "CofeeScript", "CSS" ] }
  , backend = { items = [], db = [ "Ruby", "Java", "PHP", ".Net" ] }
  }


type alias Model =
  { frontend : Langs.Model, backend : Langs.Model }


type Msg
  = Frontend Langs.Msg
  | Backend Langs.Msg


update msg model =
  case msg of
    Frontend frontendMsg ->
      { model | frontend = Langs.update frontendMsg model.frontend }

    Backend backendMsg ->
      { model | backend = Langs.update backendMsg model.backend }

```

We'll have two list of languages. Here are the defaults

The same pattern repeats, here's the main program's Model

```elm
module Main exposing (..)

import Langs
import Html exposing (..)
import Html.App exposing (beginnerProgram, map)


init =
  { frontend = { items = [], db = [ "Js", "Elm", "CofeeScript", "CSS" ] }
  , backend = { items = [], db = [ "Ruby", "Java", "PHP", ".Net" ] }
  }


type alias Model =
  { frontend : Langs.Model, backend : Langs.Model }


type Msg
  = Frontend Langs.Msg
  | Backend Langs.Msg


update msg model =
  case msg of
    Frontend frontendMsg ->
      { model | frontend = Langs.update frontendMsg model.frontend }

    Backend backendMsg ->
      { model | backend = Langs.update backendMsg model.backend }

```

We'll have two list of languages. Here are the defaults

The same pattern repeats, here's the main program's Model

Re-use the Model from the other module

```elm
module Main exposing (..)¬

import Langs
import Html exposing (..)
import Html.App exposing (beginnerProgram, map)


init =
  { frontend = { items = [], db = [ "Js", "Elm", "CofeeScript", "CSS" ] }
  , backend = { items = [], db = [ "Ruby", "Java", "PHP", ".Net" ] }
  }


type alias Model =
  { frontend : Langs.Model, backend : Langs.Model }


type Msg
  = Frontend Langs.Msg
  | Backend Langs.Msg


update msg model =
  case msg of
    Frontend frontendMsg ->
      { model | frontend = Langs.update frontendMsg model.frontend }

    Backend backendMsg ->
      { model | backend = Langs.update backendMsg model.backend }

```

We'll have two list of languages. Here are the defaults

The same pattern repeats, here's the main program's Model

Re-use the Model from the other module

The messages are coming from either one list or the other. We distinguish them by "tag"

```elm
module Main exposing (..)¬

import Langs
import Html exposing (..)
import Html.App exposing (beginnerProgram, map)


init =
  { frontend = { items = [], db = [ "Js", "Elm", "CofeeScript", "CSS" ] }
  , backend = { items = [], db = [ "Ruby", "Java", "PHP", ".Net" ] }
  }


type alias Model =
  { frontend : Langs.Model, backend : Langs.Model }


type Msg
  = Frontend Langs.Msg
  | Backend Langs.Msg


update msg model =
  case msg of
    Frontend frontendMsg ->
      { model | frontend = Langs.update frontendMsg model.frontend }

    Backend backendMsg ->
      { model | backend = Langs.update backendMsg model.backend }

```

We'll have two list of languages. Here are the defaults

The same pattern repeats, here's the main program's Model

Re-use the Model from the other module

The messages are coming from either one list or the other. We distinguish them by "tag"

We can call this a "tag"

```elm
module Main exposing (..)¬

import Langs
import Html exposing (..)
import Html.App exposing (beginnerProgram, map)


init =
  { frontend = { items = [], db = [ "Js", "Elm", "CofeeScript", "CSS" ] }
  , backend = { items = [], db = [ "Ruby", "Java", "PHP", ".Net" ] }
  }


type alias Model =
  { frontend : Langs.Model, backend : Langs.Model }


type Msg
  = Frontend Langs.Msg
  | Backend Langs.Msg


update msg model =
  case msg of
    Frontend frontendMsg ->
      { model | frontend = Langs.update frontendMsg model.frontend }

    Backend backendMsg ->
      { model | backend = Langs.update backendMsg model.backend }

```

We'll have two list of languages. Here are the defaults

The same pattern repeats, here's the main program's Model

Re-use the Model from the other module

The messages are coming from either one list or the other.
We distinguish them by "tag"

We can call this a "tag"

We delegate the update logic to the sub-module

```elm
module Main exposing (..)¬

import Langs
import Html exposing (..)
import Html.App exposing (beginnerProgram, map)


init =
  { frontend = { items = [], db = [ "Js", "Elm", "CofeeScript", "CSS" ] }
  , backend = { items = [], db = [ "Ruby", "Java", "PHP", ".Net" ] }
  }


type alias Model =
  { frontend : Langs.Model, backend : Langs.Model }


type Msg
  = Frontend Langs.Msg
  | Backend Langs.Msg


update msg model =
  case msg of
    Frontend frontendMsg ->
      { model | frontend = Langs.update frontendMsg model.frontend }

    Backend backendMsg ->
      { model | backend = Langs.update backendMsg model.backend }

```

We'll have two list of languages. Here are the defaults

The same pattern repeats, here's the main program's Model

Re-use the Model from the other module

The messages are coming from either one list or the other.
We distinguish them by "tag"

We can call this a "tag"

We delegate the update logic to the sub-module

```elm
view model =
  div
    []
    [ map Frontend (Langs.view model.frontend)
    , map Backend (Langs.view model.backend)
    ]


main =
  beginnerProgram { model = init, update = update, view = view }

```

```elm
view model =
  div
    []
    [ map Frontend (Langs.view model.frontend)
    , map Backend (Langs.view model.backend)
    ]


main =
  beginnerProgram { model = init, update = update, view = view }


```

Re-use the view from the other module

```elm
view model =
  div
    []
    [ map Frontend (Langs.view model.frontend)
    , map Backend (Langs.view model.backend)
    ]


main =
  beginnerProgram { model = init, update = update, view = view }


```

Automatically tag events coming from this view

Re-use the view from the other module

```elm
view model =
  div
    []
    [ map Frontend (Langs.view model.frontend)
    , map Backend (Langs.view model.backend)
    ]


main =
  beginnerProgram { model = init, update = update, view = view }


```

Pass only the relevant data

Automatically tag events coming from this view

Re-use the view from the other module

```elm
view model =
  div
    []
    [ map Frontend (Langs.view model.frontend)
    , map Backend (Langs.view model.backend)
    ]


main =
  beginnerProgram { model = init, update = update, view = view }


```

Pass only the relevant data

Automatically tag events coming from this view

Re-use the view from the other module

Wire the application together

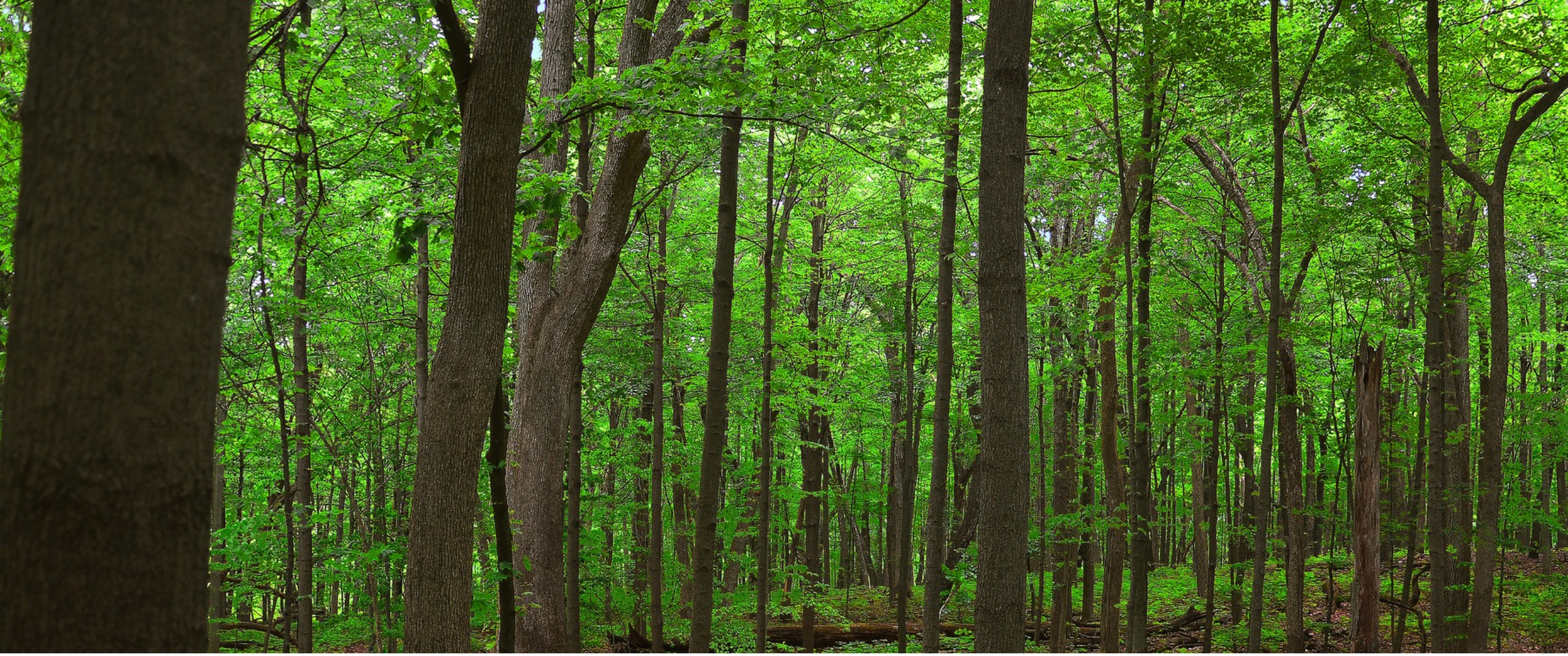
```
32 view model =
33   div
34     []
35     [ map Frontend (Langs.view model.frontend)
36     , map Backend (Langs.view model.backend)
37     ]
38
39
40 main =
41   beginnerProgram { model = init, update = update, view = view }
42
43
```

Pass only the relevant data

Automatically tag events coming from this view

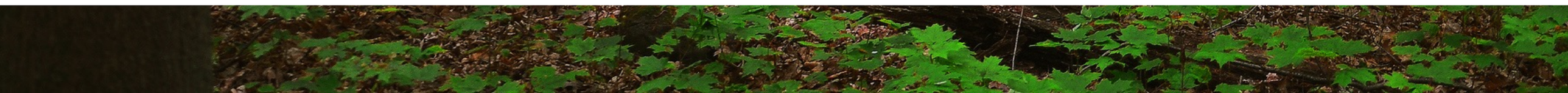
Re-use the view from the other module

Wire the application together



COMMUNITY

Elm as a community of high quality packages



Html.Lazy

Since all Elm functions are pure we have a guarantee that the same input will always result in the same output. This module gives us tools to be lazy about building `Html` that utilize this fact.

Rather than immediately applying functions to their arguments, the `lazy` functions just bundle the function and arguments up for later. When diffing the old and new virtual DOM, it checks to see if all the arguments are equal. If so, it skips calling the function!

This is a really cheap test and often makes things a lot faster, but definitely benchmark to be sure!

```
lazy : (a -> Html msg) -> a -> Html msg
```

A performance optimization that delays the building of virtual DOM nodes.

Calling `(view model)` will definitely build some virtual DOM, perhaps a lot of it. Calling `(lazy view model)` delays the call until later. During diffing, we can check to see if `model` is referentially equal to the previous value used, and if so, we just stop. No need to build up the tree structure and diff it, we know if the input to `view` is the same, the output must be the same!

THE PACKAGING SYSTEM

elm packages elm-lang / html / 1.0.0 / Html.Lazy

Html.Lazy

Since all Elm functions are pure we have a guarantee that the same input will always result in the same output. This module gives us tools to be lazy about building `Html` that utilize this fact.

Rather than immediately applying functions to their arguments, the `lazy` functions just bundle the function and arguments up for later. When diffing the old and new virtual DOM, it checks to see if all the arguments are equal. If so, it skips calling the function!

This is a really cheap test and often makes things a lot faster, but definitely benchmark to be sure!

```
lazy : (a -> Html msg) -> a -> Html msg
```

A performance optimization that delays the building of virtual DOM nodes.

Calling `(view model)` will definitely build some virtual DOM, perhaps a lot of it. Calling `(lazy view model)` delays the call until later. During diffing, we can check to see if `model` is referentially equal to the previous value used, and if so, we just stop. No need to build up the tree structure and diff it, we know if the input to `view` is the same, the output must be the same!

THE PACKAGING SYSTEM

- elm-package is the tool for sharing packages with the community

elm packages elm-lang / html / 1.0.0 / Html.Lazy

Html.Lazy

Since all Elm functions are pure we have a guarantee that the same input will always result in the same output. This module gives us tools to be lazy about building `Html` that utilize this fact.

Rather than immediately applying functions to their arguments, the `lazy` functions just bundle the function and arguments up for later. When diffing the old and new virtual DOM, it checks to see if all the arguments are equal. If so, it skips calling the function!

This is a really cheap test and often makes things a lot faster, but definitely benchmark to be sure!

```
lazy : (a -> Html msg) -> a -> Html msg
```

A performance optimization that delays the building of virtual DOM nodes.

Calling `(view model)` will definitely build some virtual DOM, perhaps a lot of it. Calling `(lazy view model)` delays the call until later. During diffing, we can check to see if `model` is referentially equal to the previous value used, and if so, we just stop. No need to build up the tree structure and diff it, we know if the input to `view` is the same, the output must be the same!

THE PACKAGING SYSTEM

- elm-package is the tool for sharing packages with the community
- Documentation in every package is automatically displayed online

elm packages elm-lang / html / 1.0.0 / Html.Lazy

Html.Lazy

Since all Elm functions are pure we have a guarantee that the same input will always result in the same output. This module gives us tools to be lazy about building `Html` that utilize this fact.

Rather than immediately applying functions to their arguments, the `lazy` functions just bundle the function and arguments up for later. When diffing the old and new virtual DOM, it checks to see if all the arguments are equal. If so, it skips calling the function!

This is a really cheap test and often makes things a lot faster, but definitely benchmark to be sure!

```
lazy : (a -> Html msg) -> a -> Html msg
```

A performance optimization that delays the building of virtual DOM nodes.

Calling `(view model)` will definitely build some virtual DOM, perhaps a lot of it. Calling `(lazy view model)` delays the call until later. During diffing, we can check to see if `model` is referentially equal to the previous value used, and if so, we just stop. No need to build up the tree structure and diff it, we know if the input to `view` is the same, the output must be the same!

THE PACKAGING SYSTEM

- elm-package is the tool for sharing packages with the community
- Documentation in every package is automatically displayed online
- The packager will prevent you from publishing broken code

elm packages elm-lang / html / 1.0.0 / Html.Lazy

Html.Lazy

Since all Elm functions are pure we have a guarantee that the same input will always result in the same output. This module gives us tools to be lazy about building `Html` that utilize this fact.

Rather than immediately applying functions to their arguments, the `lazy` functions just bundle the function and arguments up for later. When diffing the old and new virtual DOM, it checks to see if all the arguments are equal. If so, it skips calling the function!

This is a really cheap test and often makes things a lot faster, but definitely benchmark to be sure!

```
lazy : (a -> Html msg) -> a -> Html msg
```

A performance optimization that delays the building of virtual DOM nodes.

Calling `(view model)` will definitely build some virtual DOM, perhaps a lot of it. Calling `(lazy view model)` delays the call until later. During diffing, we can check to see if `model` is referentially equal to the previous value used, and if so, we just stop. No need to build up the tree structure and diff it, we know if the input to `view` is the same, the output must be the same!

THE PACKAGING SYSTEM

- elm-package is the tool for sharing packages with the community
- Documentation in every package is automatically displayed online
- The packager will prevent you from publishing broken code
- If you make an API change the packager will automatically bump the semantic version for you

PLENTY OF RESOURCES AND HELP

burak 3:52 PM
thanks @schpaencoder . can you explain a bit? it looks like it but i couldn't get the semantics behind it

schpaencoder 3:53 PM
I guess there is a (!) declaration somewhere

anyone?

krisajenkins 3:53 PM
http://package.elm-lang.org/packages/elm-lang/core/4.0.0/Platform-Cmd#!

schpaencoder 3:53 PM
`(!) : model -> List (Cmd msg) -> (model, Cmd msg)`

burak 3:54 PM
oh thanks 🙂

schnittchen 3:54 PM
I love how having just the type explains everything!

💯 3 👍 3

mordrax 3:56 PM
guys, thanks for the thought-provoking tips tonight, have to turn in, will improve for next time, thx @schpaencoder @lorenzo @dactou @glenjamin

slackbot 3:56 PM
I think you mean folks...

schpaencoder 3:56 PM
yeah, what's not to love about brevity

mordrax: 👍

dactou 3:57 PM
@mordrax: No problem!

Good luck

burak 3:52 PM
thanks @schpaencoder . can you explain a bit? it looks like it but i couldn't get the semantics behind it

schpaencoder 3:53 PM
I guess there is a (!) declaration somewhere

anyone?

krisajenkins 3:53 PM
http://package.elm-lang.org/packages/elm-lang/core/4.0.0/Platform-Cmd#!

schpaencoder 3:53 PM
```
(!) : model -> List (Cmd msg) -> (model, Cmd msg)
```

burak 3:54 PM
oh thanks 🙂

schnittchen 3:54 PM
I love how having just the type explains everything!

💯 3 👍 3

mordrax 3:56 PM
guys, thanks for the thought-provoking tips tonight, have to turn in, will improve for next time, thx @schpaencoder @lorenzo @dactou @glenjamin

slackbot 3:56 PM
I think you mean folks...

schpaencoder 3:56 PM
yeah, what's not to love about brevity

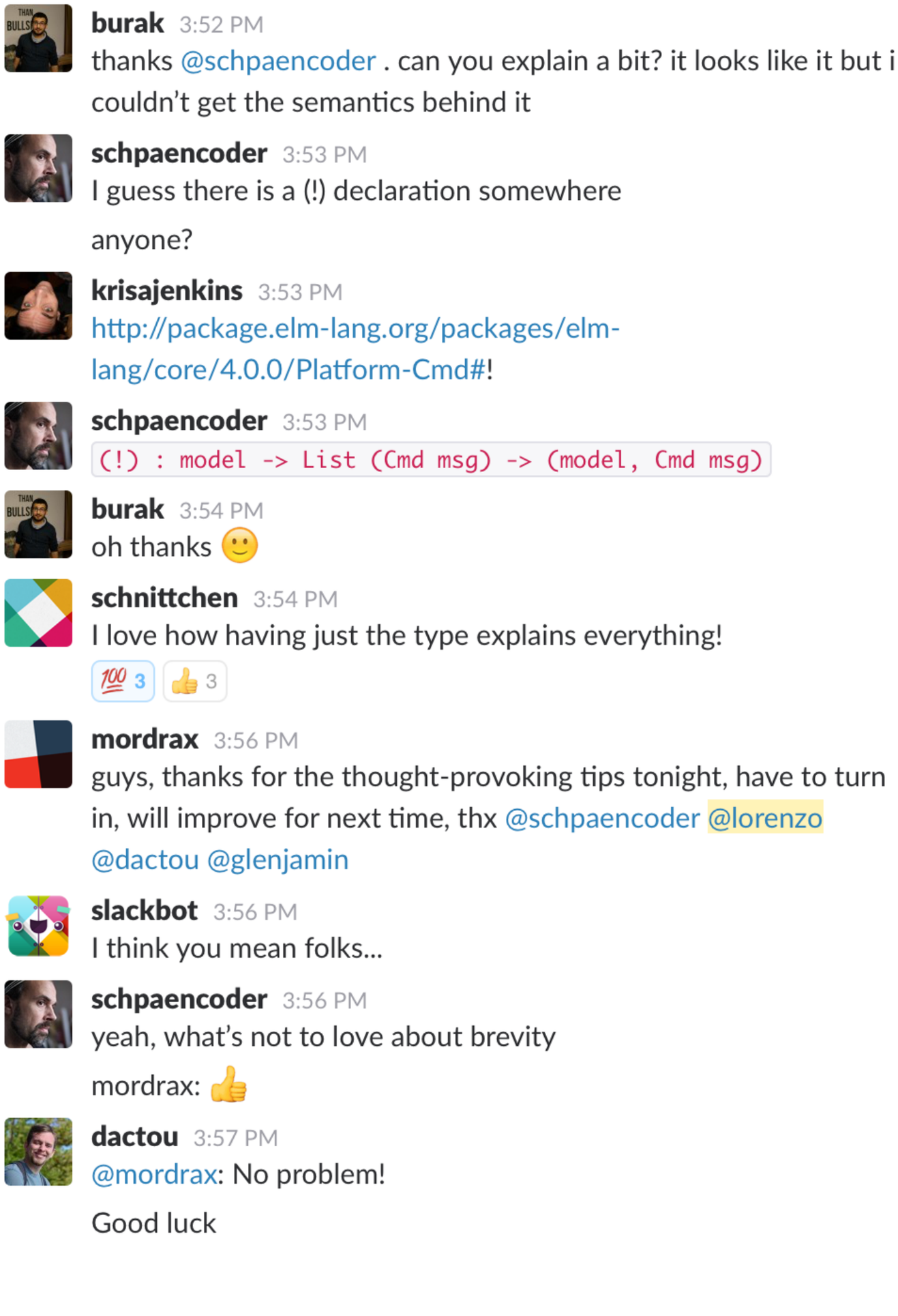
mordrax: 👍

dactou 3:57 PM
@mordrax: No problem!

Good luck

PLENTY OF RESOURCES AND HELP

- Elm was designed with usability in mind. Concepts are explained with clear examples in the guide.



PLENTY OF RESOURCES AND HELP

- Elm was designed with usability in mind. Concepts are explained with clear examples in the guide.
- Plenty of example applications linked from the homepage

burak 3:52 PM
thanks @schpaencoder . can you explain a bit? it looks like it but i couldn't get the semantics behind it

schpaencoder 3:53 PM
I guess there is a (!) declaration somewhere

anyone?

krisajenkins 3:53 PM
http://package.elm-lang.org/packages/elm-lang/core/4.0.0/Platform-Cmd#!

schpaencoder 3:53 PM
```
(!) : model -> List (Cmd msg) -> (model, Cmd msg)
```

burak 3:54 PM
oh thanks 🙂

schnittchen 3:54 PM
I love how having just the type explains everything!
💯 3 👍 3

mordrax 3:56 PM
guys, thanks for the thought-provoking tips tonight, have to turn in, will improve for next time, thx @schpaencoder @lorenzo @dactou @glenjamin

slackbot 3:56 PM
I think you mean folks...

schpaencoder 3:56 PM
yeah, what's not to love about brevity

mordrax: 👍

dactou 3:57 PM
@mordrax: No problem!

Good luck

PLENTY OF RESOURCES AND HELP

- Elm was designed with usability in mind. Concepts are explained with clear examples in the guide.
- Plenty of example applications linked from the homepage
- The best place to ask for help is the **slack channel** or IRC, where several hundred people hang out during the day.

burak 3:52 PM
thanks @schpaencoder . can you explain a bit? it looks like it but i couldn't get the semantics behind it

schpaencoder 3:53 PM
I guess there is a (!) declaration somewhere

anyone?

krisajenkins 3:53 PM
http://package.elm-lang.org/packages/elm-lang/core/4.0.0/Platform-Cmd#!

schpaencoder 3:53 PM
```
(!) : model -> List (Cmd msg) -> (model, Cmd msg)
```

burak 3:54 PM
oh thanks 🙂

schnittchen 3:54 PM
I love how having just the type explains everything!
💯 3 👍 3

mordrax 3:56 PM
guys, thanks for the thought-provoking tips tonight, have to turn in, will improve for next time, thx @schpaencoder @lorenzo @dactou @glenjamin

slackbot 3:56 PM
I think you mean folks...

schpaencoder 3:56 PM
yeah, what's not to love about brevity

mordrax: 👍

dactou 3:57 PM
@mordrax: No problem!

Good luck

PLENTY OF RESOURCES AND HELP

- Elm was designed with usability in mind. Concepts are explained with clear examples in the guide.
- Plenty of example applications linked from the homepage
- The best place to ask for help is the **slack channel** or IRC, where several hundred people hang out during the day.
- Monthly **@elmcph** meet up in Copenhagen. Next one is tomorrow!!

QUESTIONS?

QUESTIONS?

Thanks for attending!