CREATE A SIMPLE WEB APPLICATION USING ELM

> *Using Elm, I can deploy and go to sleep!*

- Mario Uher, CTO, yodel.io

WHAT IS ELM?

A functional language (Elm syntax is inspired by Haskell) which is use to create front-end web application.

Elm code will be compiled to Javascript.

WHY DO YOU SHOULD TRY ELM TODAY?

- No runtime exceptions
- Immutable values
- Small assets
- Javascript interop

NO RUNTIME EXECPTIONS

Elm uses type inference to detect corner cases and give friendly hints. Learn more

IMMUTABLE VALUES

In Elm, you cannot mutate values. The values are **immutable**. Instead of mutating values, you can create new values. So, it is easy to make a Time Traveling Debugger using Elm. Learn more

SMALL ASSETS

Elm 0.19 introdues **Dead Code Elimination** (compile with `--optimize` flag). If you use a package with hundreds of functions, like `elm/html`. Elm will automatically trims down to the 10 you actually use in your application. How does it work?

- Elm functions cannot be redefined or removed at runtime.
- Every package on package.elm-lang.org is written entirely in Elm.

Learn more

JAVASCRIPT INTEROP

You can use Elm as a small part of your existing project (Holistics). Learn more

REAL LIFE EXAMPLES

- NoRedInk
- CultureAmp

WHAT WILL YOU LEARN?

- Elm syntax and types.
- Build a simple static application and a simple stateful application.
- Handle HTTP requests and Web socket requests besides the Elm error handling technique.

TYPES

PRIMITIVES

- Int
- Float
- Char
- String
- Bool (**True** or **False**)

```
> "hello"
"hello" : String

> not True
False : Bool

> round 3.1415
3 : Int
```

LIST TYPES

In Elm, a list is a data structure for storing multiple values of the *same* kind. List is one of the most used data structures in Elm.

- List
- Tuples

```
> [ "Alice", "Bob" ]
["Alice","Bob"] : List String

> [ 1.0, 8.6, 42.1 ]
[1.0,8.6,42.1] : List Float
```

FUNCTIONS

```
> String.length
<function> : String -> Int
```

CONSTRAINTED TYPE VARIABLES

- `number` permits `Int` and `Float`
- `appendable` permits `String` and `List a`
- `comparable` permits `Int`, `Float`, `Char`, `String`, and lists/tuples of `comparable` values
- `compappend` permits `String` and `List comparable`

WRITE YOUR FIRST FUNCTION

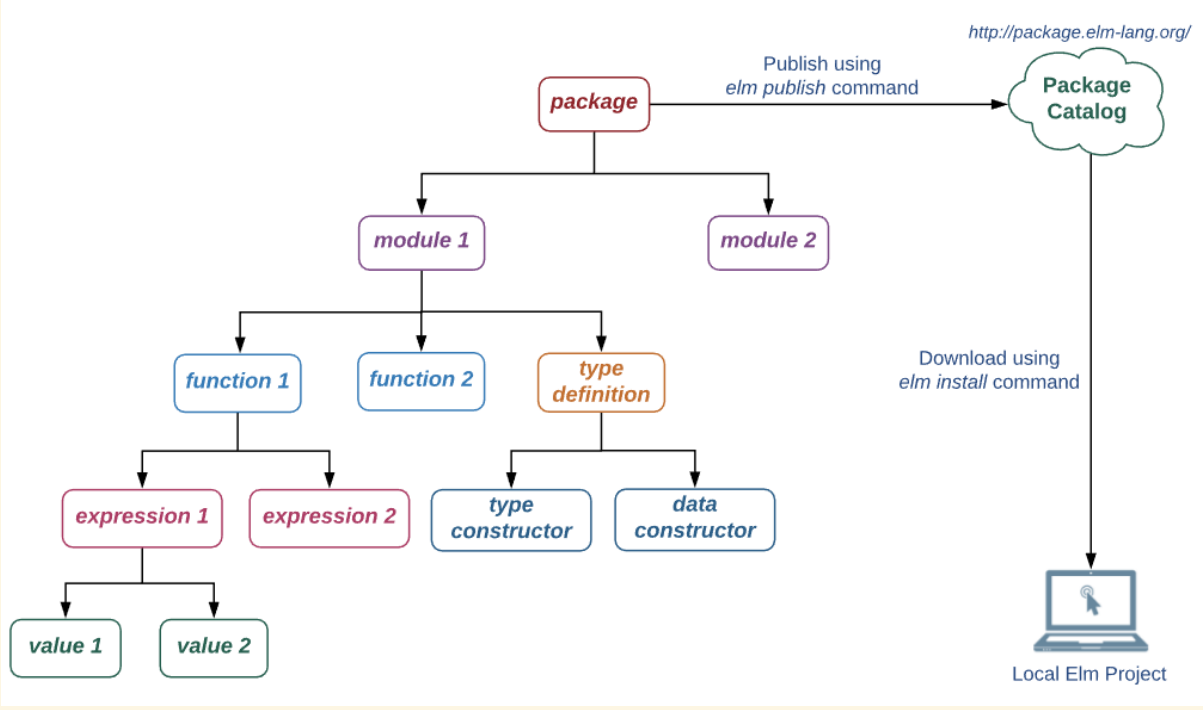
Write a function to count the number of members in a list.

Hint: (use `foldl`). Check Elm's core packages: https://package.elm-lang.org/packages/elm/core/latest

```
> List.foldl
<function> : (a -> b -> b) -> b -> List a -> b

> List.foldl (\_ b -> b + 1) 0 [3, 2, 1]
3 : number
```

WRITE YOUR FIRST STATIC WEB APPLICATION

MODULE AND PACKAGE



MODULE AND PACKAGE

Module Declaration

```
module Main exposing (main)
```

Import functions from packages

```
import Html exposing (Html, div, h1)
import Html.Attributes exposing (class, src)
```

HTML PACKAGE

Html in the normal web application

```
<div class="header">
  <h1>Hello World</h1>
<div>
```

Html defined using Elm's Html package

```
module Main exposing (main)

import Html exposing (Html, div, h1, text)
import Html.Attributes exposing (class)

main : Html msg
main =
    div [class "header"]
      [ h1 [] [text "Hello World"] ]
```

HTML PACKAGE

Check Elm Html package: https://package.elm-lang.org/packages/elm/html/latest/

DEFINE YOUR INDEX.HTML AND USING ELM COMPILED CODE

```
<body>
  <div id="main" class="main"></div>
  <script src="picshare.js"></script>
  <script>
    Elm.Picshare.init({
      node: document.getElementById('main')
    });
  </script>
</body>
```

COMPILE YOUR ELM FILE TO JAVASCRIPT

```
elm make ./src/Main.elm --output main.js
```

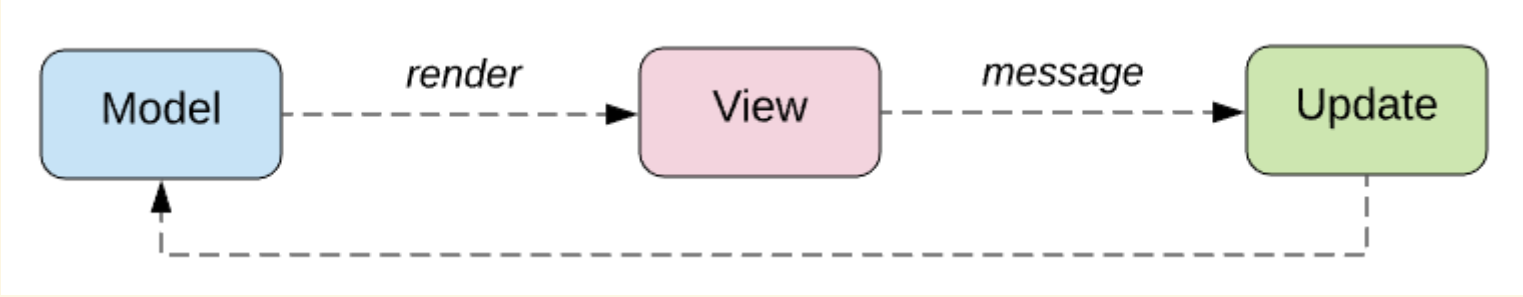
Now, let write the **Picshare** static application together.

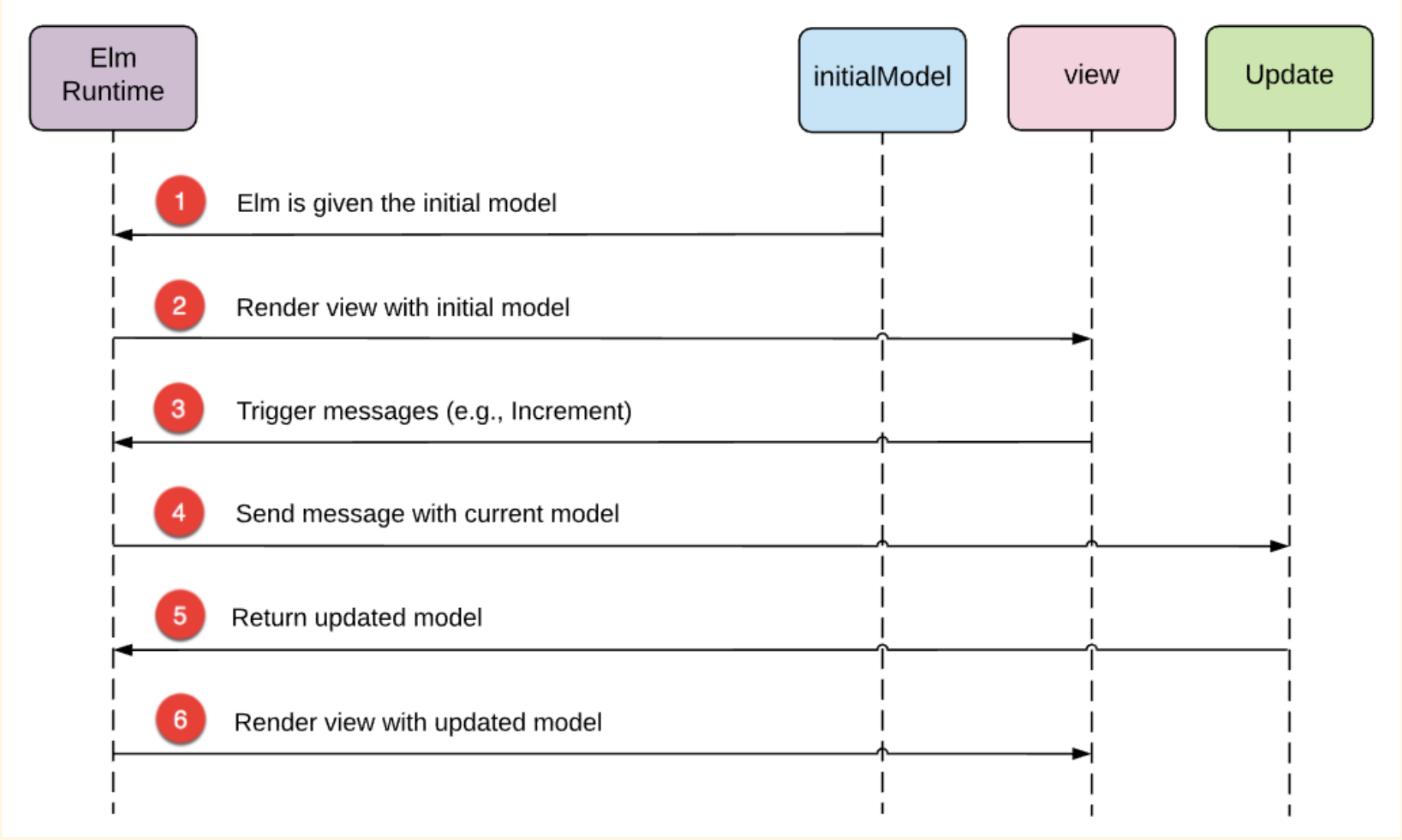
```
cd ./1-static-app
```

A STATEFUL APPLICATION

ELM ARCHITECTURE

Elm Architecture have three parts: Model, View, and Update.




Elm Runtime
initialModel
view
Update
1
Elm is given the initial model
2
Render view with initial model
3
Trigger messages (e.g., Increment)
4
Send message with current model
5
Return updated model
6
Render view with updated model

RECORD TYPE

```
> dog = { name = "Tucker", age = 11 }
{ age = 11, name = "Tucker" } : { age : number, name: String }

initialModel : { name : String, age : Int }
initialModel =
  { name = "Huy Phung"
  , age = 30
  }
```

TYPE ALIAS

```
type alias Model =
  { name : String
  , age : Int
  }

initialModel : Model
initialModel =
  { name = "Huy Phung"
  , age = 30
  }
```

TYPE VARIABLE AND CONCRETE TYPE

```
view : Html msg
view = div [] []

type Msg = Click
view : Html Msg
view = div [ onClick Click ] []
```

- The first view does not produce any events. So its type variable is not bound to anything.
- The second view produces events of your `Msg` type, and its type variable is bound to that.

Now, let's write the **Picshare** spa application together. We will:

- Use the record type, and type alias to create the initial model
- Add a love button
- Add comments

```
cd ./**2-stateful-app
```

ADD LOVE BUTTON

Picshare

file:///home/sicar/workspace/elm_workshop/2-stateful-app/index.html

Picshare

Surfing

ADD COMMENTS

Picshare

file:///home/sicar/workspace/elm_workshop/2-stateful-app/index.html

Picshare

Surfing

Comment: First comment
Comment: Second comment

Add your comment

Save

COMMUNICATE VIA HTTP

DECODER

The HTTP response is JSON data. We should not trust the data from the outside. So, we need a JSON Decoder to convert the JSON data to the valid data we use in our Elm application.

```
{
  "id": 1,
  "url": "https://programming-elm.surge.sh/1.jpg",
  "caption": "Surfing",
  "liked": false,
  "comments": ["Cowabunga, dude!"],
  "username": "surfing_usa"
}
```

DECODER

```
> import Json.Decode exposing (decodeString, bool, int, string)

> decodeString
<function>
    : Json.Decode.Decoder a -> String -> Result Json.Decode.Error
```

RESULT TYPE

The **Result** type is a built-in custom type with two constructor `Ok` and `Err`. This is how it's defined in Elm:

```
type Result error value
  = Ok value
  | Err error
```

DECODE FIELDS

```
> import Json.Decode exposing (decodeString, bool, field, int, li

> decodeString bool "true"
Ok True : Result Json.Decode.Error Bool

> decodeString string "\"Elm is Awesome\""
Ok ("Elm is Awesome") : Result Json.Decode.Error String

> decodeString (list int) "[1, 2, 3]"
Ok [1,2,3] : Result Json.Decode.Error (List Int)

> decodeString (field "name" string) """{"name": "Tucker"}"""
Ok "Tucker" : Result Json.Decode.Error String
```

DECODE AN OBJECT

```
> import Json.Decode exposing (decodeString, int, string, succeed
> import Json.Decode.Pipeline exposing (required)

required : String -> Decoder a -> Decoder (a -> b) -> Decoder b
succeed : a -> Json.Decode.Decoder a
```

DECODE AN OBJECT

```
> dog name age = { name = name, age = age }
<function> : a -> b -> { age : b, name : a }
> succeed dog
<internals> : Json.Decode.Decoder (a -> b -> { age : b, name : a

> dogDecoder =
  succeed dog
    |> required "name" string
    |> required "age" int
<internals> : Json.Decode.Decoder { age : Int, name : String }
```

WRITE THE PHOTO DECODER

```
type alias Id =
    Int

type alias Photo =
    { id : Id
    , url : String
    , caption : String
    , liked : Bool
    , comments : List String
    , newComment : String
    }

type alias Model =
```

WRITE THE PHOTO DECODER

```
photoDecoder : Decoder Photo
photoDecoder =
    succeed Photo
        |> required "id" int
        |> required "url" string
        |> required "caption" string
        |> required "liked" bool
        |> required "comments" (list string)
        |> hardcoded ""
```

TEST THE PHOTO DECODER

```
> import Picshare exposing (photoDecoder)
> import Json.Decode exposing (decodeString)
> decodeString photoDecoder """
    { "id": 1
    , "url": "https://programming-elm.surge.sh/1.jpg"
    , "caption": "Surfing"
    , "liked": false
    , "comments": ["Nước mắt em lau bằng tình yêu mới u ú ú u ù"]
    , "unknown key": "Tất nhiên là đỡ được"
    } \
    """
```

HTTP PACKAGE

Read the information of this package here: https://package.elm-lang.org/packages/elm/http/latest/Http

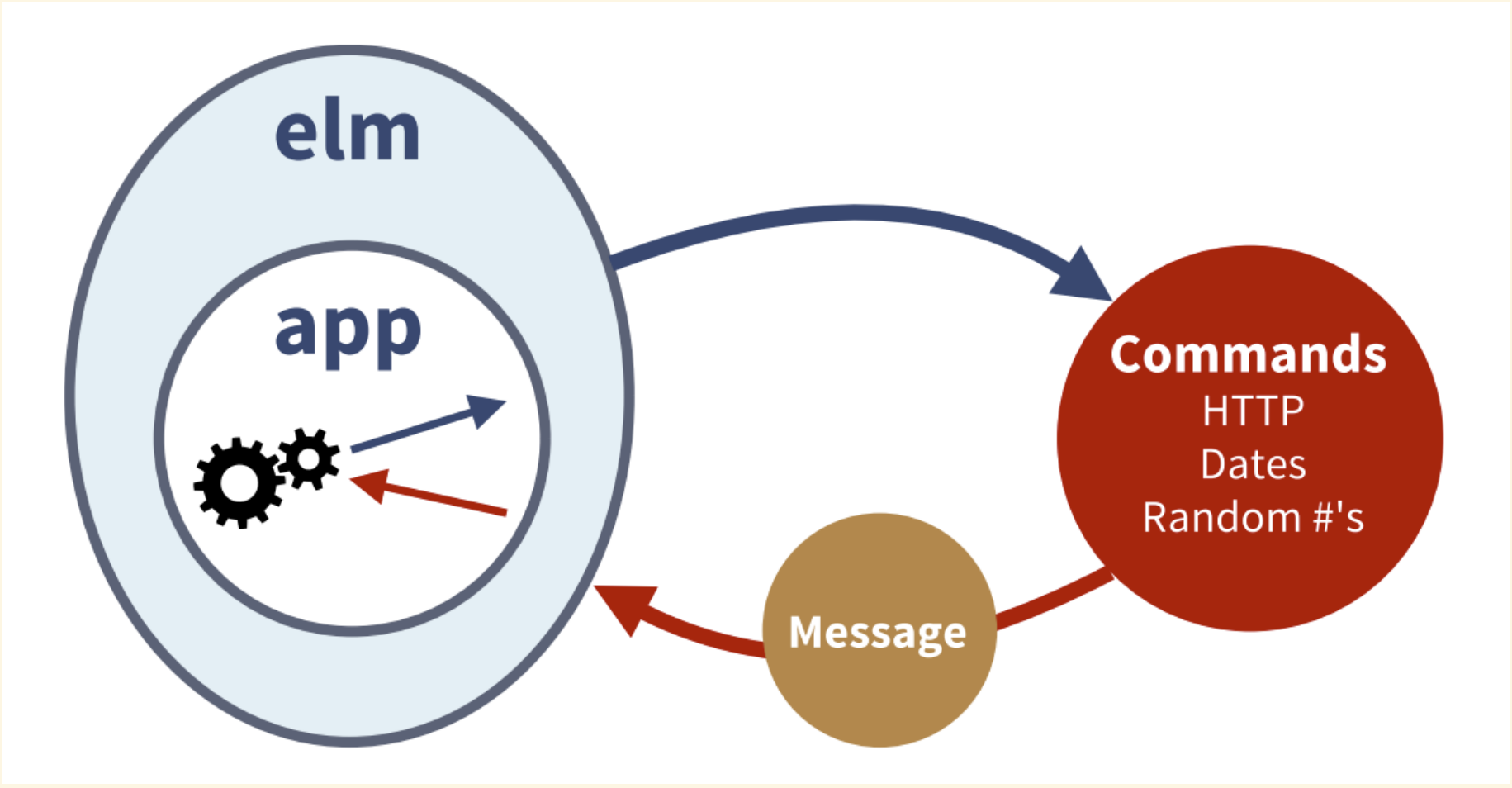
```
import Http

fetchFeed : Cmd Msg
fetchFeed =
    Http.get
      { url: baseUrl ++ "feed/1"
      , expect = Http.expectJson LoadFeed photoDecoder
      }
```

COMMAND TYPE

The **Cmd** type represents a command in Elm. Commands are special values that instruct the Elm Architecture to perform actions such as sending HTTP requests.

Elm separates creating HTTP requests from sending HTTP requests.

To communicate with the outside world, your application gives commands to the Elm Architecture. Elm will handle the command and eventually deliver the result back to your application.


elm
app
Commands
HTTP
Dates
Random #'s
Message

SAFETY HANDLE NULL USING MAYBE TYPE

The Maybe type perfectly represents a value that may or may not exist. If the value exists, then you have just that value. If the value doesn't exist, then you have nothing.

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

PRACTICE AT HOME

You could enter the folder "real-time" in the current repository. Then open this book and continue the implementation from Chapter 5.

READ MORE

VIRTUAL DOM IN ELM

Check my blog for this:
https://huyphung.one/posts/2022-03-26-virtual-dom-in-elm/

DEVELOPMENT AND DEPLOYMENT

Check my blog for this:
https://huyphung.one/posts/2022-04-14-my-setup-for-elm-application/

REFERENCES

- https://guide.elm-lang.org/
- https://elmprogramming.com/conventions-used-in-the-book.html
- https://pragprog.com/titles/jfelm/programming-elm/
- https://elm-lang.org/news/small-assets-without-the-headache
- https://huyphung.one

FACELESS VOID

PANGOLIER

NYX ASSASIN

DISRUPTOR

ARC WARDEN

URSA

TECHIES

WRAITH KING

GYROCOPTER

DARK SEER

MORPHLING

PHANTOM LANCER

SPIRIT BREAKER

LEGION COMMANDER

KUNKKA

LONE DRUID

SVEN

GRIMSTROKE