Lab 6

Web development with Elm - Introduction

Goals

In this lab you will learn to:

1. Understand the structure of Elm Web Apps
2. Create simple HTML using Elm
3. Handle side effects in a pure functional language
4. Use the `Random` module to generate random numbers
5. Write tests for the HTML generated by your application

Resources

Table 6.1: Lab Resources

Resource	Link
Elm core library	`https://package.elm-lang.org/packages/elm/core/1.0.5/`
Elm `Html` package	`https://package.elm-lang.org/packages/elm/html/latest/Html`
The `Test.Http.Query` module	`https://package.elm-lang.org/packages/elm-explorations/test/latest/Test-Html-Query`
The `Test.Http.Selector` module	`https://package.elm-lang.org/packages/elm-explorations/test/latest/Test-Html-Selector`

Table 6.2: Extra Resources - Talks about how to design Elm apps

Resource	Link
The life of a file - Evan Czaplicki	`https://youtu.be/XpDsk374LDE`
Making Impossible States Impossible - Richard Feldman	`https://youtu.be/IcgmSRJHu_8`
Immutable Relational Data - Richard Feldman	`https://youtu.be/28OdemxhfbU`

6.1 The Elm Architecture

Since in Functional Programming we can't mutate values, we need a different model to create applications than the usual MVC (Model-View-Controller), which is the de-facto design pattern used for applications with user interfaces written in imperative languages (the most prominent example being Java).

In Elm we have the Model-View-Update architecture also known as "The Elm Architecture", which maps well to Functional Programming principles.

First we have the `Model`, which is just a simple data definition (usually a record) that contains the internal state of our application.

The `View` is a function, with the signature `view : Model -> Html Msg` which returns the view displayed to the user, based on the model.

The `Update` is also a function, with the signature `update: Msg -> Model -> Model` which takes a *message* (action), the current `model` and returns a new `model`.

The `Msg` type is defined to contain all the possible actions the user can perform that change the state of the application.

These actions are dispatched by the `View` returned from the `view` function, and trigger the `udpate` function, which updates the state (`model`) and then the view is re-rendered. This is all managed by the Elm Runtime.

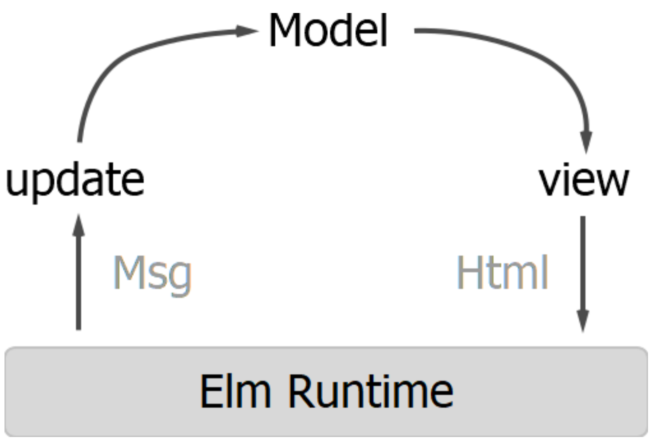


Figure 6.1: The Elm Architecture

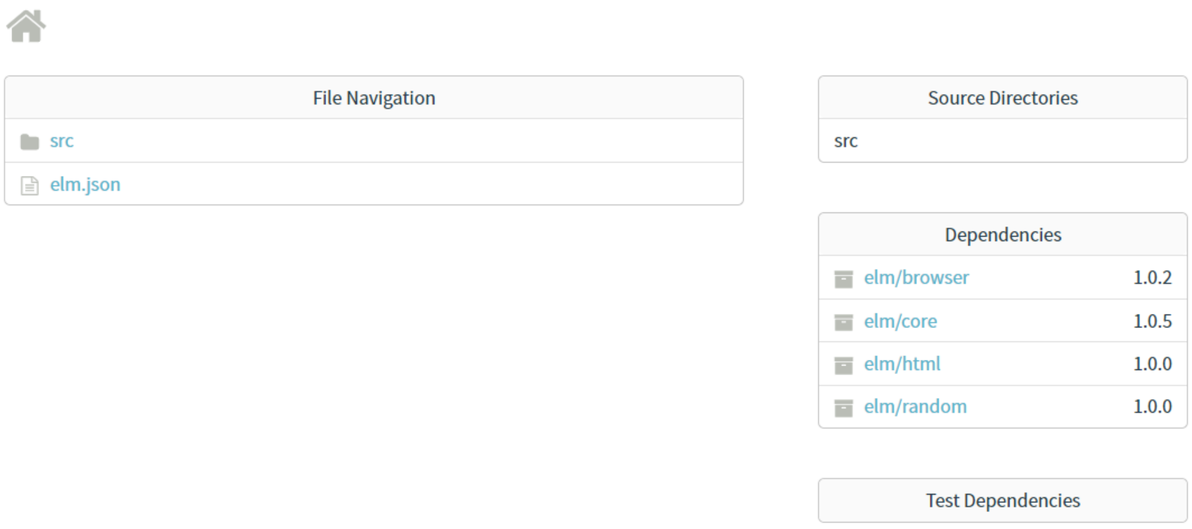
6.2 Elm Web Apps by example

6.2.1 Developing Elm Web Apps

To "run" an Elm Web App, simply open a shell in the folder where your `elm.json` file is located and type `elm reactor`:

```
                                              powershell session
PS > elm reactor
Go to http://localhost:8000 to see your project dashboard.
```

Then you can open the displayed link to view your project dashboard. To open an app, navigate to the `src` folder and click on the corresponding source file. You should then see your app.



File Navigation

src
elm.json

Source Directories

src

Dependencies

elm/browser	1.0.2
elm/core	1.0.5
elm/html	1.0.0
elm/random	1.0.0

Test Dependencies

Figure 6.2: Elm Reactor dashboard

After you make any changes to the source files, simply reload the browser page (press F5) to see the updates reflected in the app. Note that the app will revert to the initial state and the current session will be lost.

6.2.2 Simple HTML

Listing 6.2.1 of Recipe.elm — Elm code

```elm
module Recipe exposing (..)

import Html exposing (..)
import Html.Attributes exposing (..)

main = div []
  [ h1 [ style "font-family" "arial" ]
    [ a
      [ href """http://www.jamieoliver.com/recipes/chocolate-recipes/
                bloomin-brilliant-brownies"""]
      [ text "Bloomin’ brilliant brownies" ]
    ]
  , h3 [] [ i [] [ text "Ingredients:" ]]
  , ul []
    [ li [] [ text "200 g quality dark chocolate (70%)"]
    , li [] [ text "250 g unsalted butter"]
    , li [] [ text "75 g dried sour cherries , optional"]
    , li [] [ text "50 g chopped nuts , optional"]
    , li [] [ text "80 g quality cocoa powder"]
    , li [] [ text "65 g plain flour"]
    , li [] [ text "1 teaspoon baking powder"]
    , li [] [ text "360 g caster sugar"]
    , li [] [ text "4 large free-range eggs"]
    ]
  , h3 [] [ i [] [ text "Method:" ]]
  , p [] [ text """Preheat the oven to 180C/350F/gas 4.
                   Line a 24cm square baking tin with greaseproof paper."""]
  , p [] [ text """Snap the chocolate into a large bowl,
                   add the butter and place over a pan of simmering water,
                   until melted, stirring regularly.
                   Stir through the cherries and nuts (if using)."""]
  , p [] [ text """Sift the cocoa powder and flour into a separate bowl,
                   add the baking powder and sugar, then mix together."""]
  , p [] [ text """Add the dry ingredients to the chocolate,
                   cherry and nut mixture and stir together well. Beat the eggs,
                   then mix in until you have a silky consistency."""]
  , p [] [ text """Pour the brownie mix into the baking tin,
                   and place in the oven for around 25 minutes.
                   You don’t want to overcook them so, unlike cakes, you don’t want
                   a skewer to come out clean - the brownies should be slightly
                   springy on the outside but still gooey in the middle."""]
  ]
```

The first example isn’t really an app, it’s just a plain, simple HTML page, but it illustrates how we can create HTML views using Elm.

Here we created a page for a brownie recipe using simple HTML tags like `div`, `h1`, `i` (italics), `ul` (unordered list) and `li` (list item), created by the functions with the same name. The main pattern that should notice is that aside from `text` each function takes as arguments 2 lists: the first list contains the *attributes* of the element and the second list the *children*. The `text` function simply return plaintext in the HTML DOM.

Exercise 6.2.1 *

Starting from the code above and the type definition for `Recipe`, write a function `recipeView : Recipe -> Html msg` that can render any recipe (i.e. avoid hardcoding the recipe data into the view)

Listing 6.2.2: Recipe Elm code

```elm
type alias Recipe =
  { title: String
  , linkToOriginal: String
  , ingredients: List String
  , method: List String
  }
```

Hint:

- Do you know the number of ingredients or steps in advance?
- What is common for all elements in the ingredients list?
- What is common for all elements that represent steps?

6.2.3 Counter app

Listing 6.2.4 of `Counter.elm` Elm code

```elm
module Counter exposing (..)

import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)
import Html.Attributes exposing (style)
import Html.Attributes exposing (disabled)

main =
  Browser.sandbox { init = 0, update = update, view = view }

type alias Model = Int

type Msg = Increment | Decrement

update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view : Model -> Html Msg
view model =
  let
    bigFont = style "font-size" "20pt"
  in
    div []
      [ button [ bigFont, onClick Increment ] [ text "+" ]
      , div [ bigFont ] [ text (String.fromInt model) ]
      , button [ bigFont, onClick Decrement ] [ text "-" ]
      ]
```

`main`

As you've seen in other programming languages (C, C++ and Java) each program must have a `main` function which acts as the *entry point*.

Elm apps are no different, and here we specify the 3 functions that we need to build our app:

- `init`, the function that generates the starting state
- `update`, the function that handles the actions the user can take by updating the model
- `view`, the function which takes the model and returns the HTML that will be shown to the user

We use the `Browser.sandbox` function to "build" the app from the `init`, `update` and `view` functions.

`Model`

For the counter app the state can be simply represented as an `Int`, so defining a type alias is enough.

`Msg`

In this application we can `Increment` or `Decrement` the value of the counter, so we define a union type which represents these actions.

`update`

The `update` function takes the action performed by the user (of type `Msg`), the current state (of type `Model`) and returns the next state. In this case if `msg` is `Increment`, the counter is incremented (i.e. we return `model + 1`) and if `msg` is `Decrement`, the counter is decremented (i.e. we return `model - 1`).

`view`

The `view` function takes the `Model` and generates `Html` based on it. Here we generate 2 buttons and text that displays the current value of the counter. We use the `onClick` function to set the click handlers for the buttons, which will dispatch the appropriate action (message) (`Increment` or `Decrement`) for each button.

Exercise 6.2.2 *

Modify the Counter app to prevent the counter from going over 10 or under -10 by disabling the + or - buttons when the value is reached.
Remove the call to `skip` in the `CounterTests.elm` file to test your implementation.

Hint:
Use the `disabled`[a] attribute.

[a] https://package.elm-lang.org/packages/elm/html/latest/Html-Attributes#disabled

Exercise 6.2.3 **

Modify the Counter app to make the text red when the counter is close (is greater than 8 or less than -8) to 10 or -10.

Hint:

- You can use the `style`[a] attribute.
- First, just try to add the attribute so that one of the text is always red.
- Think about the difference between the `disabled` and the `style` attributes.
  - How did you "enable" or "disable" the `disabled` attribute based on a condition?
  - Can you do the same for `style`?
  - If not, how can you make its presence optional?

[a] https://package.elm-lang.org/packages/elm/html/latest/Html-Attributes#style

6.3 Handling side effects

Lets reflect a bit on the code we've seen so far:

In the first example, we had one function, which just returned a page with all the data hard-coded. We then rewrote the code to separate the data from the page rendering logic.

In the second example we again structured the app in such a way to separate the data and logic, but this time it could also handle user input, in a similar fashion to a *state machine*: after we provided an input, the app transitioned to a new state the depended only on the current state and the input.

To make useful apps, our needs to able to interact with more than just the user: it needs to able to interact with the browser to obtain things like random numbers and the current time and, it needs to be able to interact with servers to retrieve and store data, and it also needs to able to interact with JavaScript code to use browser APIs that Elm doesn't expose. From this point, we will name the browser, JavaScript and servers collectively as the "outside world".

At this point you might have a question: "How can we say that the app is isolated from the outside world if it can react when we click on the button?". The answer is simple: The Elm runtime could handle these actions completely independently.

Contrast that with getting the current time. In the case of web apps, the current time comes from the browser, which in turn obtains the time from the operating system it's running on, which in turn relies on the hardware and the internet to be able to keep time accurately.

The same applies to requesting data from servers: from the moment the request is sent to the moment the data arrives back a **lot** can go wrong: the server might be down, the data we requested might not be available, it might be changed in ways we don't expect, it might arrive very slowly and so on.

To solve this problem, Elm introduces the concept of *commands*: "tasks" that we give to the Elm runtime that deal with these aspects (randomness, getting the time, getting or sending data and interacting with JavaScript). These tasks are given to the Elm runtime in the update function as commands and when the Elm runtime completes the task it sends us a message with the result of the task (which might succeed or fail).

6.3.1 Obtaining random values: Coinflip app

main

Listing 6.3.1 of `CoinFlip.elm (main)` Elm code

```elm
main =
  Browser.element
    { init = init
    , update = update
    , subscriptions = subscriptions
    , view = view
    }
```

In the counter app, we used the `Browser.sandbox` function to create our app, which *isolates* our app from the outside world. The only events it could receive are the interactions of the user with the buttons.

Now we will use the `Browser.element` function, which allows interacting with the outside world and takes slightly more complex version of the `init` and `update` functions to allow interacting with the outside world, and it additionally takes a `subscriptions` function which you can ignore for now.

`Model` and `init`

Listing 6.3.2 of `CoinFlip.elm (Coin, Model, init)` `Elm code`

```elm
type CoinSide
  = Heads
  | Tails
type alias Model =
  { currentFlip : Maybe CoinSide
  , flips: List CoinSide
  }
init : () -> (Model, Cmd Msg)
init _ =
  ( initModel
  , Cmd.none
  )
```

Here we defined the type for our `CoinSide` which can be `Heads` or `Tails`. The `Model` will consist of the current flip, which will initially be `None` and a list of previous flips.

Also note that now the `init` function returns a tuple consisting of the initial state and a command to the Elm runtime which will be executed when the app is started.

`Msg` and `update`

Listing 6.3.3 of `CoinFlip.elm (Msg, update)` `Elm code`

```elm
type Msg
  = Flip
  | AddFlip CoinSide
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Flip ->
      ( model
      , Random.generate AddFlip coinFlip
      )

    AddFlip coin ->
      ( Model (Just coin) (coin::model.flips)
      , Cmd.none
      )
```

This is perhaps the most interesting function of this app, because compared to the `udpate` function of the counter app, it returns a tuple, with the new `Model` and a *command* of type `Cmd` for the Elm runtime (just like the `init` function above). As we discussed above, commands can be used to interact with the outside world, like making HTTP requests, getting the current time or generating random numbers (our use case).

Since these actions can't be dispatched directly from the view (the view dispatches only messages), the view simply sends a normal message which won't cause any change to the model, but will send a message to the Elm runtime to execute the given task.

`Random.generate` takes 2 arguments: a constructor function which will be used to dispatch an update when the task completed (in our case the random number is generated) and a random `Generator` instance, like `Int` which generates a random number in a given range or `uniform`, which takes a list as an argument and returns each value with equal probability.

Generating a random coin: `coinFlip`

Listing 6.3.4 of `CoinFlip.elm (coinFlip)` Elm code

```elm
coinFlip : Random.Generator CoinSide
coinFlip =
  Random.uniform Heads
    [ Tails ]
```

Here the most important aspect to note is that the `Random.uniform` function takes 2 parameters: one element and a list of elements. This is because it doesn't make sense to randomly sample an empty list (what should we return in this case?).

`view`

Listing 6.3.5 of `CoinFlip.elm (view, coinToString, viewCoin)` Elm code

```elm
view : Model -> Html Msg
view model =
  let
    currentFlip =
      model.currentFlip
      |> Maybe.map viewCoin
      |> Maybe.withDefault (text "Press the flip button to get started")
    flips =
      model.flips
      |> List.map coinToString
      |> List.intersperse " "
      |> List.map text
  in
    div []
      [ button [ onClick Flip ] [ text "Flip" ]
      , currentFlip
      , div [] flips
      ]
coinToString : CoinSide -> String
coinToString coin =
  case coin of
    Heads -> "h"
    Tails -> "t"
viewCoin : CoinSide -> Html Msg
viewCoin coin =
  let
    name = coinToString coin
  in
    div [ style "font-size" "4em" ] [ text name ]
```

For the `view` function, the important aspects to note is the organization:

1. We use 2 functions, `coinToString` and `viewCoin` to generate a `String` and to generate

HTML from a coin instance.

2. We use these 2 functions to generate the view for the current flip and the list of previous flips.

3. We use pipelines to make the sequence of transformations a given value goes through more clear.

6.4 Testing the generated HTML

In section 5.7 on page 78 we saw how we can write tests to check if our function work correctly.

Since the function for generating the HTML is pure (it only depends on the model), testing that the output conforms to our expectations is easy.

To get started, besides our usual imports for testing, we also need to import the `Test.Html.Query` and `Test.Html.Selector` modules. We'll use qualified imports to be able to refer to their exports more concisely.

Listing 6.4.1: Imports for testing HTML — Elm code

```elm
import Expect exposing (Expectation)
import Test exposing (..)

import Test.Html.Query as Q
import Test.Html.Selector as S
import Html.Attributes as Attr
```

The `Test.Html.Query` module exports function that allow us to find *nodes* in the HTML DOM and make assertions about these nodes, while `Test.Html.Selector` provides the predicates that will be used for finding nodes and the assertions.

Lets take a look at an example, which we will use to explain the details:

Listing 6.4.2 of `CounterTests.elm (viewHasTwoButtons)` — Elm code

```elm
viewHasTwoButtons : Test
viewHasTwoButtons =
    test "view contains two buttons" <|
        \_ ->
            CounterClass.view 0
                |> Q.fromHtml
                |> Q.findAll [ S.tag "button" ]
                |> Q.count (Expect.equal 2)
```

The example begins like any other test, using the `test` function. Since we want to test the generated HTML, the value returned by the `view` function will be our starting point. The first step in the pipeline is to make it *queryable* by using the `fromHtml` function. We then search for all the elements that *match a selector* using the `findAll` function. We will find the buttons by their tags, using the `tag` function. Finally we will check how many nodes were found using the `count` function, which takes as parameter an expectation. The other way to "end" a query is to use the `has` function, which checks if there is at least one element that matches the given selectors:

Listing 6.4.3 of CounterTests.elm (viewContainsTheCurrentCount) Elm code

```elm
viewContainsTheCurrentCount : Test
viewContainsTheCurrentCount =
    test "view contains the current count" <|
        \_ ->
            CounterClass.view 0
                |> Q.fromHtml
                |> Q.has [ S.text (String.fromInt 0) ]
```

Exercise 6.4.1 *

Write a test for the coin flip app to test that the initial view contains the text “Press the flip button to get started”.

Hint:
Use the `initModel` function to create the initial, empty model.

6.4.1 Using `id` and `class` to make it easier to find elements

You may know, that in order to style a HTML page, we can use CSS (Cascading Style Sheets). The most frequently used HTML attribute to define and then apply styles to a variety of elements is `class`. To define a unique name for a given element, you can use `id`.

In the context of testing, we can use it to “give names” to elements, which will be present in the generated HTML and thus we can use them during testing to find certain elements.

For example we would like to test that when rendering a recipe there is always at least one ingredient. For this we can search for the unordered list that contains the ingredients and then select its children to count them:

Listing 6.4.4 of RecipeTests.elm (atLeastOneIngredient) Elm code

```elm
atLeastOneIngredient : Test
atLeastOneIngredient = test "each recipe contains at least one ingredient" <|
  \_ ->
    RecipeView.recipeView RecipeView.brownies
      |> Q.fromHtml
      |> Q.find [S.tag "ul"]
      |> Q.children [S.tag "li"]
      |> Q.count (Expect.atLeast 1)
```

This code is reasonable, but think about the case when you might have multiple lists, that are in turn nested in some other elements.

The solution to easily select all ingredients, is to add the `class "ingredient"` attribute to each ingredient. Then we can select the ingredients regardless of where they are in the document:

Listing 6.4.5 of `RecipeTests.elm (atLeastOneIngredientClass)` — `Elm code`

```elm
atLeastOneIngredientClass : Test
atLeastOneIngredientClass =
  skip <| test "each recipe contains at least one ingredient (using classes)" <|
    \_ ->
      RecipeView.recipeView RecipeView.brownies
        |> Q.fromHtml
        |> Q.findAll [S.class "ingredient"]
        |> Q.count (Expect.atLeast 1)
```

Exercise 6.4.2 *

Change your solution to Exercise 6.2.1 by adding the `ingredient` class to the view for each ingredient, such that the `atLeastOneIngredientClass` and `eachIngredientHasClassIngredient` both pass.

6.5 Review questions

Question 6.5.1 *

What are the 3 components of the Elm Architecture?

Question 6.5.2 *

What is the fundamental difference between a command and a message?

Hint:
Who is the recipient of each item? Which (if any) is always needed for interactive apps?

Question 6.5.3 *

Which are the two steps of a command?

6.6 Practice problems

Exercise 6.6.1 *

Modify the Coin flip app to display the number of heads and tails outcomes so far, in two ways:

1. Keep the number in the `Model` and simply display it in the `view`
2. Compute the values from the `flips` field of the `Model` each time in the `view`

Exercise 6.6.2 **

Modify the Coin flip app to have an initial flip (i.e. when the user loads the app, it should initially display heads or tails instead of current message).

Hint:

Try to answer the following questions before writing the code:

- What function will you modify to solve this problem?
- Can you make the `Model` simpler after making this change?

Exercise 6.6.3 **

Add a “Flip 10” and a “Flip 100” button to the Coin flip app that triggers 10 and 100 coin flips respectively.

Hint:
You can implement this in 2 different ways:

- By using the `Cmd.batch` function. Hacky for this particular problem, but you don’t need to change the logic in other places.
- By using the `Random.list` function. The “correct” method, but you’ll have to make changes to the `AddFlip` variant of the `Msg` type to hold a list of coin flips.

In both cases you should update the `Flip` variant of the `Msg` type to hold the number of coin flips to be performed.