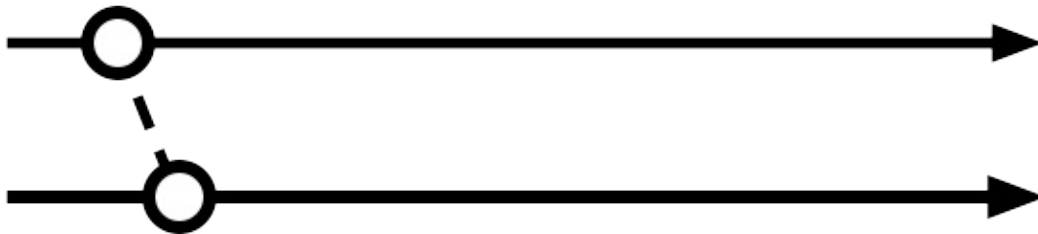

Table of Contents

Introduction	1.1
Foundations	1.2
Hello World	1.2.1
Functions	1.2.2
More on functions	1.2.3
Imports and modules	1.2.4
Union types	1.2.5
Type aliases	1.2.6
The unit type	1.2.7
The Elm architecture	1.3
Introduction	1.3.1
Structure	1.3.2
Messages	1.3.3
Application flow	1.3.4
Messages with payload	1.3.5
Subscriptions and commands	1.4
Subscriptions	1.4.1
Commands	1.4.2
Starting an application	1.5
Planning	1.5.1
Backend	1.5.2
Webpack 1	1.5.3
Webpack 2	1.5.4
Webpack 3	1.5.5
Webpack 4	1.5.6
Multiple modules	1.5.7
Resources	1.6
Intro	1.6.1
Models	1.6.2
Main	1.6.3
Players list	1.6.4
Main view	1.6.5
Fetching	1.7
Plan	1.7.1
Messages	1.7.2
Models	1.7.3
Commands	1.7.4
Update	1.7.5
Views	1.7.6

Try it	1.7.7
Routing	1.8
Intro	1.8.1
Routing	1.8.2
Player edit view	1.8.3
Models	1.8.4
Messages	1.8.5
Update	1.8.6
View	1.8.7
Main	1.8.8
Try it	1.8.9
Navigation	1.8.10
Try it	1.8.11
Editing	1.9
Plan	1.9.1
Messages	1.9.2
Player edit	1.9.3
Commands	1.9.4
Update	1.9.5
Conclusion	1.10
Improvements	1.10.1
Tips and Tricks	1.11
Context	1.11.1
Point free style	1.11.2
Composing	1.11.3
Composing - Parent	1.11.4
Composing - Flow	1.11.5
Troubleshooting	1.11.6



```
learnElm =  
  List.map read elmTutorial
```

Elm Tutorial

A tutorial on developing single page web applications (SPAs) with [Elm](#).

This tutorial covers:

- Some Elm foundations
- Understanding commands and subscriptions in Elm
- Understanding the Elm architecture
- Breaking an application in sub components and resources
- Integrating CSS
- Fetching and parsing JSON
- Routing
- CRUD operations

Read it online [here](#).

You can also download offline version [here](#) (PDF, ePub, Mobi).

Code

Code for the example application built in the second part of this tutorial can be found at <https://github.com/sporto/elm-tutorial-app>.

Requirements

For this tutorial you will need:

- Elm version 0.18 (Installation is covered later in the tutorial)
- Node JS version 4 +
- Yarn package manager <https://yarnpkg.com/en/>

Contributing

Please open issues and send PRs at <https://github.com/sporto/elm-tutorial>.

[Share on Twitter](#) | [Follow @sebasporto](#)



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

© Sebastian Porto 2016

Foundations

This chapter covers:

- Running a basic Elm application
- Basic of functions and types in Elm

Hello World

Installing Elm

Go to <http://elm-lang.org/install> and download the appropriate installer for your system.

Our first Elm application

Let's write our first Elm application. Create a folder for your application. In this folder run the following command in the terminal:

```
elm package install elm-lang/html
```

This will inform you that additional packages are needed, show you the proposed upgrade plan, and ask you to confirm the upgrade plan. If you are running elm 0.18.0, the upgrade plan will include elm-lang/core, elm-lang/html, and elm-lang/virtual-dom packages.

This will create an *elm-package.json* file and *elm-stuff* directory in the same project directory, and then install the specified modules. You modules themselves are saved in the *elm-stuff* directory, while their specifications are saved in the *elm-package.json* file.

Add a `Hello.elm` file, with the following code:

```
module Hello exposing (..)

import Html exposing (text)

main =
    text "Hello"
```

Go to this folder on the terminal and type:

```
elm reactor
```

This should show you:

```
elm reactor 0.18.0
Listening on http://0.0.0.0:8000/
```

Open `http://0.0.0.0:8000/` on a browser. And click on `Hello.elm`. You should see `Hello` on your browser.

Note to you might see a warning about a missing type annotation for `main`. Ignore this for now, we will get to type annotations later.

Let's review what is happening here:

Module declaration

```
module Hello exposing (..)
```

Every module in Elm must start with a module declaration, in this case the module name is called `Hello`. It is a convention to name the file and the module the same e.g. `Hello.elm` contains `module Hello`.

The `exposing (..)` part of the declaration specifies what function and types this module exposes to the other modules importing this. In this case we expose everything `(..)`.

Imports

```
import Html exposing (text)
```

In Elm you need to import the **modules** you want to use explicitly. In this case we want to use the **Html** module.

This module has many functions to work with html. We will be using `.text` so we import this function into the current namespace by using `exposing`.

Main

```
main =  
  text "Hello"
```

Front end applications in Elm start on a function called `main`. `main` is a function that returns an element to draw into the page. In this case it returns an Html element (created by using `text`).

Elm reactor

Elm **reactor** creates a server that compiles Elm code on the fly. **reactor** is useful for developing applications without worrying too much about setting up a build process. However **reactor** has limitations, so we will need to switch to a build process later on.

Function basics

This chapter covers basic Elm syntax that is important to get familiar with: functions, function signatures, partial application and the pipe operator.

Functions

Elm supports two kind of functions:

- anonymous functions
- named functions

Anonymous function

An anonymous function, as its name implies, is a function we create without a name:

```
\x -> x + 1

\x y -> x + y
```

Between the backslash and the arrow, you list the arguments of the function, and on the right of the arrow, you say what to do with those arguments.

Named functions

A named function in Elm looks like this:

```
add1 : Int -> Int
add1 x =
  x + 1
```

- The first line in the example is the function signature. This signature is optional in Elm, but recommended because it makes the intention of your function clearer.
- The rest is the implementation of the function. The implementation must follow the signature defined above.

In this case the signature is saying: Given an integer (Int) as argument return another integer.

You call it like:

```
add1 3
```

In Elm we use *whitespace* to denote function application (instead of using parenthesis).

Here is another named function:

```
add : Int -> Int -> Int
add x y =
  x + y
```

This function takes two arguments (both Int) and returns another Int. You call this function like:

```
add 2 3
```


No arguments

A function that takes no arguments is a constant in Elm:

```
name =  
  "Sam"
```

How functions are applied

As shown above a function that takes two arguments may look like:

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

We can think of this signature as a function that takes two floats and returns another float:

```
divide 5 2 == 2.5
```

However, this is not quite true, in Elm all functions take exactly one argument and return a result. This result can be another function. Let's explain this using the function above.

```
-- When we do:  
  
divide 5 2  
  
-- This is evaluated as:  
  
((divide 5) 2)  
  
-- First `divide 5` is evaluated.  
-- The argument `5` is applied to `divide`, resulting in an intermediate function.  
  
divide 5 -- -> intermediate function  
  
-- Let's call this intermediate function `divide5`.  
-- If we could see the signature and body of this intermediate function, it would look like:  
  
divide5 : Float -> Float  
divide5 y =  
  5 / y  
  
-- So we have a function that has the `5` already applied.  
  
-- Then the next argument is applied i.e. `2`  
  
divide5 2  
  
-- And this returns the final result
```

The reason we can avoid writing the parenthesis is because function application **associates to the left**.

Grouping with parentheses

When you want to call a function with the result of another function call you need to use parentheses for grouping the calls:

```
add 1 (divide 12 3)
```

Here the result of `divide 12 3` is given to `add` as the second parameter.

In contrast, in many other languages it would be written:

```
add(1, divide(12, 3))
```

Partial application

As explained above every function takes only one argument and returns another function or a result. This means you can call a function like `add` above with only one argument, e.g. `add 2` and get a *partially applied function** back. This resulting function has a signature `Int -> Int`.

`add 2` returns another function with the value `2` bound as the first parameter. Calling the returned function with a second value returns `2 +` the second value:

```
add2 = add 2
add2 3 -- result 5
```

Partial application is incredibly useful in Elm for making your code more readable and passing state between functions in your application.

The pipe operator

As shown above you can nest functions like:

```
add 1 (multiply 2 3)
```

This is a trivial example, but consider a more complex example:

```
sum (filter (isOver 100) (map getCost records))
```

This code is difficult to read, because it resolves inside out. The pipe operator allows us to write such expressions in a more readable way:

```
3
  |> multiply 2
  |> add 1
```

This relies heavily on partial application as explained before. In this example the value `3` is passed to a partially applied function `multiply 2`. Its result is in turn passed to another partially applied function `add 1`.

Using the pipe operator the complex example above would be written like:

```
records
  |> map getCost
  |> filter (isOver 100)
  |> sum
```

More on functions

Type variables

Consider a function with a type signature like:

```
indexOf : String -> List String -> Int
```

This hypothetical function takes a string and a list of strings and returns the index where the given string was found in the list or -1 if not found.

But what if we instead have a list of integers? We wouldn't be able to use this function. However, we can make this function **generic** by using **type variables** or **stand-ins** instead of specific types.

```
indexOf : a -> List a -> Int
```

By replacing `String` with `a`, the signature now says that `indexOf` takes a value of any type `a` and a list of that same type `a` and returns an integer. As long as the types match the compiler will be happy. You can call `indexOf` with a `String` and a list of `String`, or an `Int` and a list of `Int`, and it will work.

This way functions can be made more generic. You can have several **type variables** as well:

```
switch : ( a, b ) -> ( b, a )  
switch ( x, y ) =  
  ( y, x )
```

This function takes a tuple of types `a`, `b` and returns a tuple of types `b`, `a`. All these are valid calls:

```
switch (1, 2)  
switch ("A", 2)  
switch (1, ["B"])
```

Note that any lowercase identifier can be used for type variables, `a` and `b` are just a common convention. For example the following signature is perfectly valid:

```
indexOf : thing -> List thing -> Int
```

Functions as arguments

Consider a signature like:

```
map : (Int -> String) -> List Int -> List String
```

This function:

- takes a function: the `(Int -> String)` part
- a list of integers
- and returns a list of strings

The interesting part is the `(Int -> String)` fragment. This says that a function must be given conforming to the `(Int -> String)` signature.

For example, `toString` from core is such function. So you could call this `map` function like:

```
map toString [1, 2, 3]
```

But `Int` and `String` are too specific. So most of the time you will see signatures using stand-ins instead:

```
map : (a -> b) -> List a -> List b
```

This function maps a list of `a` to a list of `b`. We don't really care what `a` and `b` represent as long as the given function in the first argument uses the same types.

For example, given functions with these signatures:

```
convertStringToInt : String -> Int  
convertIntToString : Int -> String  
convertBoolToInt : Bool -> Int
```

We can call the generic map like:

```
map convertStringToInt ["Hello", "1"]  
map convertIntToString [1, 2]  
map convertBoolToInt [True, False]
```

Imports and modules

In Elm you import a module by using the `import` keyword e.g.

```
import Html
```

This imports the `Html` module. Then you can use functions and types from this module by using its fully qualified path:

```
Html.div [] []
```

You can also import a module and expose specific functions and types from it:

```
import Html exposing (div)
```

`div` is mixed in the current scope. So you can use it directly:

```
div [] []
```

You can even expose everything in a module:

```
import Html exposing (..)
```

Then you would be able to use every function and type in that module directly. But this is not recommended most of the time because we end up with ambiguity and possible clashes between modules.

Modules and types with the same name

Many modules export types with the same name as the module. For example, the `Html` module has an `Html` type and the `Task` module has a `Task` type.

So this function that returns an `Html` element:

```
import Html

myFunction : Html.Html
myFunction =
  ...
```

Is equivalent to:

```
import Html exposing (Html)

myFunction : Html
myFunction =
  ...
```

In the first one we only import the `Html` module and use the fully qualified path `Html.Html`.

In the second one we expose the `Html` type from the `Html` module. And use the `Html` type directly.

Module declarations

When you create a module in Elm, you add the `module` declaration at the top:

```
module Main exposing (..)
```

`Main` is the name of the module. `exposing (..)` means that you want to expose all functions and types in this module. Elm expects to find this module in a file called **Main.elm**, i.e. a file with the same name as the module.

You can have deeper file structures in an application. For example, the file **Players/Utils.elm** should have the declaration:

```
module Players.Utils exposing (..)
```

You will be able to import this module from anywhere in your application by:

```
import Players.Utils
```

Union types

In Elm, **Union Types** are used for many things as they are incredibly flexible. A union type looks like:

```
type Answer = Yes | No
```

`Answer` can be either `Yes` or `No`. Union types are useful for making our code more generic. For example a function that is declared like this:

```
respond : Answer -> String
respond answer =
  ...
```

Can either take `Yes` or `No` as the first argument e.g. `respond Yes` is a valid call.

Union types are also commonly called **tags** in Elm.

Payload

Union types can have associated information with them:

```
type Answer = Yes | No | Other String
```

In this case, the tag `other` will have an associated string. You could call `respond` like this:

```
respond (other "Hello")
```

You need the parenthesis otherwise Elm will interpret this as passing two arguments to `respond`.

As constructor functions

Note how we add a payload to `other`:

```
other "Hello"
```

This is just like a function call where `other` is the function. Union types behave just like functions. For example given a type:

```
type Answer = Message Int String
```

You will create a `Message` tag by:

```
Message 1 "Hello"
```

You can do partial application just like any other function. These are commonly called `constructors` because you can use this to construct complete types, i.e. use `Message` as a function to construct `(Message 1 "Hello")`.

Nesting

It is very common to 'nest' one union type in another.

```
type OtherAnswer = DontKnow | Perhaps | Undecided

type Answer = Yes | No | Other OtherAnswer
```

Then you can pass this to our `respond` function (which expect `Answer`) like this:

```
respond (Other Perhaps)
```

Type variables

It is also possible to use type variables or stand-ins:

```
type Answer a = Yes | No | Other a
```

This is an `Answer` that can be used with different types, e.g. `Int`, `String`.

For example, `respond` could look like this:

```
respond : Answer Int -> String
respond answer =
  ...
```

Here we are saying that the `a` stand-in should be of type `Int` by using the `Answer Int` signature.

So later we will be able to call `respond` with:

```
respond (Other 123)
```

But `respond (Other "Hello")` would fail because `respond` expects an integer in place of `a` .

A common use

One typical use of union types is passing around values in our program where the value can be one of a known set of possible values.

For example, in a typical web application, we can trigger messages to perform actions, e.g. load users, add user, delete user, etc. Some of these messages would have a payload.

It is common to use union types for this:

```
type Msg
  = LoadUsers
  | AddUser
  | EditUser UserId
  ...
```

There is a lot more about Union types. If interested read more about this [here](#).

Type aliases

A **type alias** in Elm is, as its name says, an alias for something else. For example, in Elm you have the core `Int` and `String` types. You can create aliases for them:

```
type alias PlayerId = Int

type alias PlayerName = String
```

Here we have created a couple of type alias that simply point to other core types. This is useful because instead of having a function like:

```
label: Int -> String
```

You can write it like:

```
label: PlayerId -> PlayerName
```

In this way, it is much clearer what the function is asking for.

Records

A record definition in Elm looks like:

```
{ id : Int
, name : String
}
```

If you were to have a function that takes a record, you would have to write a signature like:

```
label: { id : Int, name : String } -> String
```

Quite verbose, but type aliases help a lot with this:

```
type alias Player =
  { id : Int
  , name : String
  }

label: Player -> String
```

Here we create a `Player` type alias that points to a record definition. Then we use that type alias in our function signature.

Constructors

Type aliases can be used as **constructor** functions. Meaning that we can create a real record by using the type alias as a function.

```
type alias Player =  
  { id : Int  
    , name : String  
  }  
  
Player 1 "Sam"  
==> { id = 1, name = "Sam" }
```

Here we create a `Player` type alias. Then, we call `Player` as a function with two parameters. This gives us back a record with the proper attributes. Note that the order of the arguments determines which values will be assigned to which attributes.

The unit type

The empty tuple `()` is called the **unit type** in Elm. It is so prevalent that it deserves some explanation.

Consider a type alias with a **type variable** (represented by `a`):

```
type alias Message a =  
  { code : String  
  , body : a  
  }
```

You can make a function that expects a `Message` with the `body` as a `String` like this:

```
readMessage : Message String -> String  
readMessage message =  
  ...
```

Or a function that expects a `Message` with the `body` as a List of Integers:

```
readMessage : Message (List Int) -> String  
readMessage message =  
  ...
```

But what about a function that doesn't need a value in the body? We use the unit type for indicating that the body should be empty:

```
readMessage : Message () -> String  
readMessage message =  
  ...
```

This function takes `Message` with an **empty body**. This is not the same as **any value**, just an **empty** one.

So the unit type is commonly used as a placeholder for an empty value.

Task

A real world example of this is the `Task` type. When using `Task`, you will see the unit type very often.

A typical task has an **error** and a **result**:

```
Task error result
```

- Sometimes we want a task where the error can be safely ignored: `Task () result`
- Or the result is ignored: `Task error ()`
- Or both: `Task () ()`

The Elm architecture

This chapter covers:

- Overview of the Elm architecture
- Introduction to `Html.program`
- Messages
- Commands
- Subscriptions

This page covers Elm 0.18

Introduction

When building front end applications in Elm, we use the pattern known as the Elm architecture. This pattern provides a way of creating self contained components that can be reused, combined, and composed in endless variety.

Elm provides the `Html` module for this. This is easier to understand by building a small app.

Install elm-html:

```
elm package install elm-lang/html
```

Create a file called **App.elm**:

```
module App exposing (..)

import Html exposing (Html, div, text, program)

-- MODEL

type alias Model =
    String

init : ( Model, Cmd Msg )
init =
    ( "Hello", Cmd.none )

-- MESSAGES

type Msg
    = NoOp

-- VIEW

view : Model -> Html Msg
view model =
    div []
        [ text model ]

-- UPDATE

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        NoOp ->
            ( model, Cmd.none )

-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none

-- MAIN

main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

You can run this program running:

```
elm reactor
```

And opening <http://localhost:8000/App.elm>

This is a lot of code for just showing "Hello", but it will help us understand the structure of even very complex Elm applications.

This page covers Elm 0.18

Structure of Html.program

Imports

```
import Html exposing (Html, div, text, program)
```

- We will use the `Html` type from the `Html` module, plus a couple of functions `div`, `text` and `program`.

Model

```
type alias Model =  
    String  
  
init : ( Model, Cmd Msg )  
init =  
    ( "Hello", Cmd.none )
```

- First we define our application model as a type alias, in this kind. Here it is just a `String`.
- Then we define an `init` function. This function provides the initial input for the application.

Html.program expects a tuple with `(model, command)`. The first element in this tuple is our initial state, e.g. "Hello". The second element is an initial command to run. More on this later.

When using the elm architecture, we compose all components models into a single state tree. More on this later too.

Messages

```
type Msg  
    = NoOp
```

Messages are things that happen in our applications that our component responds to. In this case, the application doesn't do anything, so we only have a message called `NoOp`.

Other examples of messages could be `Expand` or `Collapse` to show and hide a widget. We use union types for messages:

```
type Msg  
    = Expand  
    | Collapse
```

View

```
view : Model -> Html Msg  
view model =  
    div []  
        [ text model ]
```

The function `view` renders an Html element using our application model. Note that the type signature is `Html Msg`. This means that this Html element would produce messages tagged with `Msg`. We will see this when we introduce some interaction.

Update


```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    NoOp ->
      ( model, Cmd.none )
```

Next we define an `update` function, this function will be called by `Html.program` each time a message is received. This update function responds to messages updating the model and returning commands as needed.

In this example, we only respond to `NoOp` and return the unchanged model and `Cmd.none` (meaning no command to perform).

Subscriptions

```
subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.none
```

We use subscriptions to listen for external input to our application. Some examples of subscriptions are:

- Mouse movements
- Keyboard events
- Browser location changes

In this case, we are not interested in any external input so we use `Sub.none` .

Main

```
main : Program Never Model Msg
main =
  program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

Finally `Html.program` wires everything together and returns an html element that we can render in the page. `program` takes our `init` , `view` , `update` and `subscriptions` .

This page covers Elm 0.18

Messages

In the last section, we created an application using `Html.program` that was just static `Html`. Now let's create an application that responds to user interaction using messages.

```
module Main exposing (..)

import Html exposing (Html, button, div, text, program)
import Html.Events exposing (onClick)

-- MODEL

type alias Model =
    Bool

init : ( Model, Cmd Msg )
init =
    ( False, Cmd.none )

-- MESSAGES

type Msg
    = Expand
    | Collapse

-- VIEW

view : Model -> Html Msg
view model =
    if model then
        div []
            [ button [ onClick Collapse ] [ text "Collapse" ]
            , text "Widget"
            ]
    else
        div []
            [ button [ onClick Expand ] [ text "Expand" ] ]

-- UPDATE

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Expand ->
            ( True, Cmd.none )

        Collapse ->
            ( False, Cmd.none )

-- SUBSCRIPTIONS
```

```
subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none

-- MAIN

main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

This program is very similar to the previous program we did, but now we have two messages: `Expand` and `Collapse`. You can run this program by copying it into a file and opening it using Elm reactor.

Let's look more closely at the `view` and `update` functions.

View

```
view : Model -> Html Msg
view model =
    if model then
        div []
            [ button [ onClick Collapse ] [ text "Collapse" ]
            , text "Widget"
            ]
    else
        div []
            [ button [ onClick Expand ] [ text "Expand" ] ]
```

Depending on the state of the model we show either the collapsed or the expanded view.

Note the `onClick` function. As this view is of type `Html Msg` we can trigger messages of that type using `onClick`. `Collapse` and `Expand` are both of type `Msg`.

Update

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Expand ->
            ( True, Cmd.none )

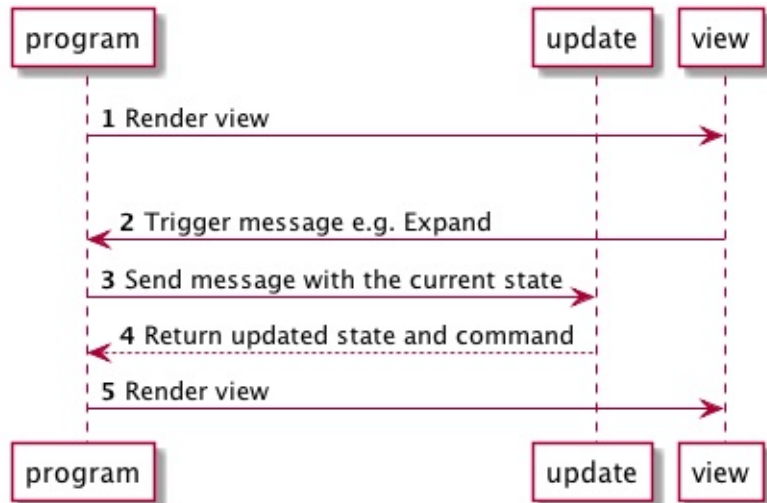
        Collapse ->
            ( False, Cmd.none )
```

`update` responds to the possible messages. Depending on the message, it returns the desired state. When the message is `Expand`, the new state will be `True` (expanded).

Next let's see how **Html.program** orchestrates these pieces together.

Application flow

The following diagram illustrates how the pieces of our application interact with `Html.program`.



1. `Html.program` calls our view function with the initial model and renders it.
2. When the user clicks on the Expand button, the view triggers the `Expand` message.
3. `Html.program` receives the `Expand` message which calls our `update` function with `Expand` and the current application state.
4. The update function responds to the message by returning the updated state and a command to run (or `Cmd.none`).
5. `Html.program` receives the updated state, stores it, and calls the view with the updated state.

Usually `Html.program` is the only place where an Elm application holds state, it is centralised in one big state tree.

This page covers Elm 0.18

Messages with payload

You can send a payload in your message:

```
module Main exposing (..)

import Html exposing (Html, button, div, text, program)
import Html.Events exposing (onClick)

-- MODEL

type alias Model =
    Int

init : ( Model, Cmd Msg )
init =
    ( 0, Cmd.none )

-- MESSAGES

type Msg
    = Increment Int

-- VIEW

view : Model -> Html Msg
view model =
    div []
        [ button [ onClick (Increment 2) ] [ text "+" ]
        , text (toString model)
        ]

-- UPDATE

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Increment howMuch ->
            ( model + howMuch, Cmd.none )

-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none

-- MAIN

main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

Note how the `Increment` message requires an integer:

```
type Msg
  = Increment Int
```

Then in the view we trigger that message with a payload:

```
onClick (Increment 2)
```

And finally in update we use **pattern matching** to extract the payload:

```
update msg model =
  case msg of
    Increment howMuch ->
      ( model + howMuch, Cmd.none )
```

Subscriptions and Commands

In order to listen to external input and create **side effects** in our application we need to learn about **Subscriptions** and **Commands**.

This chapter covers:

- Subscriptions
- Commands

This page covers Elm 0.18

Subscriptions

In Elm, using **subscriptions** is how your application can listen for external input. Some examples are:

- [Keyboard events](#)
- [Mouse movements](#)
- Browser locations changes
- [Websocket events](#)

To illustrate this, let's create an application that responds to both keyboard and mouse events.

First, install the required libraries:

```
elm package install elm-lang/mouse
elm package install elm-lang/keyboard
```

Then, create this program:

```
module Main exposing (..)

import Html exposing (Html, div, text, program)
import Mouse
import Keyboard

-- MODEL

type alias Model =
    Int

init : ( Model, Cmd Msg )
init =
    ( 0, Cmd.none )

-- MESSAGES

type Msg
    = MouseMsg Mouse.Position
    | KeyMsg Keyboard.KeyCode

-- VIEW

view : Model -> Html Msg
view model =
    div []
        [ text (toString model) ]

-- UPDATE

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        MouseMsg position ->
```

```
( model + 1, Cmd.none )

KeyMsg code ->
  ( model + 2, Cmd.none )

-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.batch
    [ Mouse.clicks MouseMsg
    , Keyboard.downs KeyMsg
    ]

-- MAIN

main : Program Never Model Msg
main =
  program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

Run this program with Elm reactor. Each time you click the mouse you will see the counter increasing by one; each time you press a key you will see the counter increasing by 2.

Let's review the important parts relevant to subscriptions in this program.

Messages

```
type Msg
  = MouseMsg Mouse.Position
  | KeyMsg Keyboard.KeyCode
```

We have two possible messages: `MouseMsg` and `KeyMsg`. These will trigger when the mouse or the keyboard are pressed, accordingly.

Update

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    MouseMsg position ->
      ( model + 1, Cmd.none )

    KeyMsg code ->
      ( model + 2, Cmd.none )
```

Our update function responds to each message differently, so it increases the counter by one when we press the mouse or by two when we press a key.

Subscriptions

```
subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.batch ❸
    [ Mouse.clicks MouseMsg ❶
      , Keyboard.downs KeyMsg ❷
    ]
```

Here we declare the things we want to listen to. We want to listen to `Mouse.clicks` ❶ and `Keyboard.downs` ❷. Both of these functions take a message constructor and return a subscription.

We use `Sub.batch` ❸ so we can listen to both of them. `batch` takes a list of subscriptions and returns one subscription which includes all of them.

This page covers Elm 0.18

Commands

In Elm, commands (`Cmd`) are how we tell the runtime to execute things that involve side effects. For example:

- Generate a random number
- Make an http request
- Save something into local storage

A `Cmd` can be one or a collection of things to do. We use commands to gather all the things that need to happen and hand them to the runtime. Then the runtime will execute them and feed the results back to the application.

In other words, every function returns a value in a functional language such as Elm. Function side effects in the traditional sense are forbidden by the language design and Elm takes an alternative approach to modeling them. Essentially, a function returns a command value which names the desired effect. Due to the Elm architecture, the main `Html.program` we've been using is the ultimate recipient of this command value. The update method of the `Html.program` then contains the logic to execute the named command.

Let's try an example app using commands:

```
module Main exposing (..)

import Html exposing (Html, div, button, text, program)
import Html.Events exposing (onClick)
import Random

-- MODEL

type alias Model =
    Int

init : ( Model, Cmd Msg )
init =
    ( 1, Cmd.none )

-- MESSAGES

type Msg
    = Roll
    | OnResult Int

-- VIEW

view : Model -> Html Msg
view model =
    div []
        [ button [ onClick Roll ] [ text "Roll" ]
        , text (toString model)
        ]

-- UPDATE

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Roll ->
            ( model, Random.generate OnResult (Random.int 1 6) )

        OnResult res ->
            ( res, Cmd.none )

-- MAIN

main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = (always Sub.none)
        }
```

If you run this application it will show a button that will generate a random number each time you click it.

Let's review the relevant parts:

Messages

```
type Msg
  = Roll
  | OnResult Int
```

We have two possible messages in our application. `Roll` for rolling a new number. `OnResult` for getting a generated number back from the `Random` library.

Update

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    Roll ->
      ( model, Random.generate① OnResult (Random.int 1 6) )

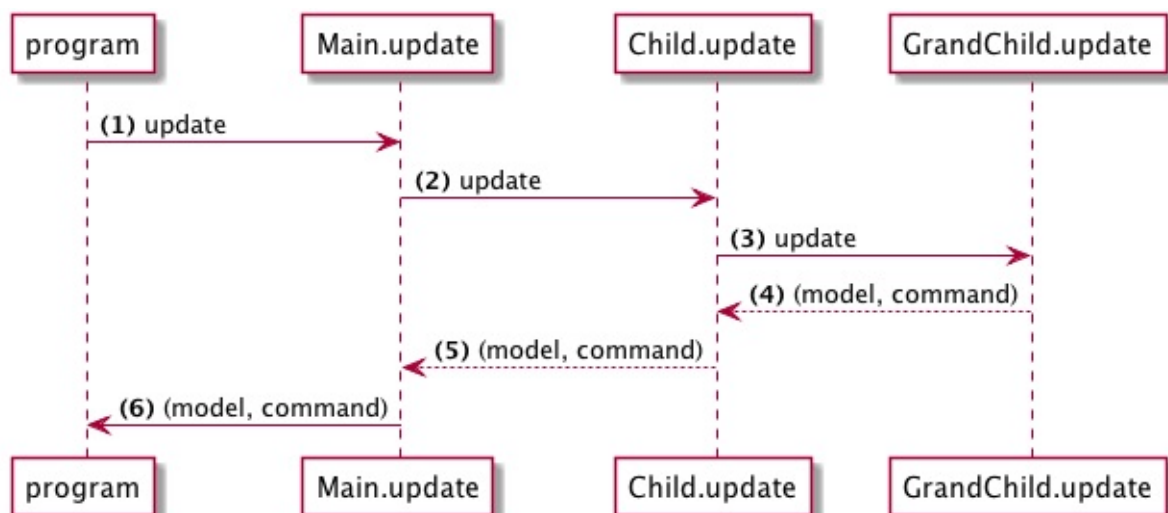
    OnResult res ->
      ( res, Cmd.none )
```

① `Random.generate` creates a command that will generate random numbers. This function requires the first argument to be a constructor for the message that will be fed back to our application. In this case our constructor is `OnResult`.

So when the command is run Elm will call `OnResult` with the generated number, producing `OnResult 2` for example. Then **Html.program** will feed this message back to application.

In case you're wondering, `OnResult res` denotes a message, `OnResult`, containing an additional payload of information, the Integer 'res' in this case. This pattern is known as parameterized types.

In a bigger application with many nested components we can potentially send many commands at once to **Html.program**. Take this diagram for example:



Here we collect commands from three different levels. At the end we send all these commands to **program** to run.

This page covers Tutorial v2. Elm 0.18.

Starting an application

In this chapter we start building an example Elm application.

This tutorial will cover the following aspects about building an application:

- Application structure
- Fetching resources
- Views
- Routing
- User interaction and saving changes

See next page for details about the application.

This page covers Tutorial v2. Elm 0.18.

Planning

We will build a basic application to track an imaginary role playing game.

Resources

During the rest of this guide I will use the word **resources** to refer to models that are the subject of our application. These are **players** in this application. Using the word **model** can be confusing because component specific state is also a model (for example the expanded / collapse state of a component).

Wireframes

The application will have two views:

The wireframes show two views of the application. View 1, titled 'Players', displays a list of four players with their names and levels, and an 'Edit' button for each. View 2, titled 'Player', shows the details for 'Player 1', including their name and level, with buttons to increment or decrement the level and a 'Done' button.

1		Players	
Player 1	3	<button>Edit</button>	
Player 2	2	<button>Edit</button>	
Player 3	1	<button>Edit</button>	
Player 4	2	<button>Edit</button>	

2		Player	
Name	Player 1		
Level	2	<button>-</button>	<button>+</button>
<button>Done</button>			

Screen 1

Will show a list of players. From here you can:

- Navigate to edit a player

Screen 2

Shows the edit view for a player. In this screen you can:

- Change the level

This is a very simple application that will demonstrate:

- Multiple views
- Nested components
- Breaking the application into resources
- Routing
- Shared state across the application
- Read and edit operation on the records
- Ajax requests

This page covers Tutorial v2. Elm 0.18.

Backend

We will need a backend for our application, we can use **json-server** for this.

json-server is an npm package that provides a quick way to create fake APIs.

Start a new node project:

```
yarn init
```

Accept all the defaults.

Install **json-server**:

```
yarn add json-server@0.9.5
```

Make **api.js** in the root of the project:

```
var jsonServer = require('json-server')

// Returns an Express server
var server = jsonServer.create()

// Set default middlewares (logger, static, cors and no-cache)
server.use(jsonServer.defaults())

var router = jsonServer.router('db.json')
server.use(router)

console.log('Listening at 4000')
server.listen(4000)
```

Add **db.json** at the root:

```
{
  "players": [
    { "id": "1", "name": "Sally", "level": 2 },
    { "id": "2", "name": "Lance", "level": 1 },
    { "id": "3", "name": "Aki", "level": 3 },
    { "id": "4", "name": "Maria", "level": 4 },
    { "id": "5", "name": "Julian", "level": 1 },
    { "id": "6", "name": "Jaime", "level": 1 }
  ]
}
```

Start the server by running:

```
node api.js
```

Test this fake API by browsing to:

- <http://localhost:4000/players>

This page covers Tutorial v2. Elm 0.18.

Webpack

Elm reactor is great for prototyping simple applications, but for a bigger app it falls short. As it is now, **reactor** doesn't support talking with external JavaScript or importing external CSS. To overcome these issues we will use **Webpack** to compile our Elm code instead of Elm reactor.

Webpack is a code bundler. It looks at your dependency tree and only bundles the code that is imported. Webpack can also import CSS and other assets inside a bundle. Read more about Webpack [here](#).

There are many alternatives that you can use to achieve the same as Webpack, for example:

- [Browserify](#)
- [Gulp](#)
- [StealJS](#)
- [JSPM](#)
- Or if using a framework like Rails or Phoenix you can bundle the Elm code and CSS using them.

Requirements

You will need Node JS version 4 or more for these libraries to work as expected.

Installing webpack and loaders

Install webpack and associated packages:

```
yarn add webpack webpack-dev-middleware webpack-dev-server elm-webpack-loader file-loader style-loader css-loader url-loader
```

This tutorial is using **webpack** version **1.13** and **elm-webpack-loader** version **4.1**.

Loaders are extensions that allow webpack to load different formats. E.g. `css-loader` allows webpack to load .css files.

We also want to use a couple of extra libraries:

- [Basscss](#) for CSS, `ace-css` is the Npm package that bundles common Basscss styles
- [FontAwesome](#) for icons

```
yarn add ace-css@1.1 font-awesome@4
```

Webpack config

We need to add a **webpack.config.js** at the root:

```
var path = require("path");

module.exports = {
  entry: {
    app: [
      './src/index.js'
    ]
  },

  output: {
    path: path.resolve(__dirname + '/dist'),
    filename: '[name].js',
  },

  module: {
    rules: [
      {
        test: /\.css|scss$/,
        use: [
          'style-loader',
          'css-loader',
        ]
      },
      {
        test: /\.html$/,
        exclude: /node_modules/,
        loader: 'file-loader?name=[name].[ext]',
      },
      {
        test: /\.elm$/,
        exclude: [/elm-stuff/, /node_modules/],
        loader: 'elm-webpack-loader?verbose=true&warn=true',
      },
      {
        test: /\.woff(2)?(\?v=[0-9]\.[0-9]\.[0-9])?$/,
        loader: 'url-loader?limit=10000&mimetype=application/font-woff',
      },
      {
        test: /\.ttf|eot|svg)(\?v=[0-9]\.[0-9]\.[0-9])?$/,
        loader: 'file-loader',
      },
    ],
  },

  noParse: /\.elm$/,
},

devServer: {
  inline: true,
  stats: { colors: true },
},
};
```

Things to note:

- This config creates a Webpack dev server, see the key `devServer` . We will be using this server for development instead of Elm reactor.
- Entry point for our application will be `./src/index.js` , see the `entry` key.

This page covers Tutorial v2. Elm 0.18.

Webpack 2

index.html

As we are not using Elm reactor anymore we will need to create our own HTML for containing the application. Create **src/index.html**:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Elm SPA example</title>
  </head>
  <body>
    <div id="main"></div>
    <script src="/app.js"></script>
  </body>
</html>
```

index.js

This is the entry point that Webpack will look for when creating a bundle. Add **src/index.js**:

```
'use strict';

require('ace-css/css/ace.css');
require('font-awesome/css/font-awesome.css');

// Require index.html so it gets copied to dist
require('./index.html');

var Elm = require('./Main.elm');
var mountNode = document.getElementById('main');

// .embed() can take an optional second argument. This would be an object describing the data we need to start a program, i.e. a userID or some token
var app = Elm.Main.embed(mountNode);
```

Install Elm packages

Run:

```
elm package install elm-lang/html
```

After doing this there should be a file called **elm-package.json** in the root of your project.

Source directory

We will be adding all our source code in the `src` folder, so we need to tell Elm where to search for dependencies. In **elm-package.json** change:

```
...  
"source-directories": [  
  "src"  
],  
...
```

Without this the Elm compiler will try to find the imports in the root of our project and fail.

This page covers Tutorial v2. Elm 0.18.

Webpack 3

Initial Elm app

Create a basic Elm app. In **src/Main.elm**:

```
module Main exposing (..)

import Html exposing (Html, div, text, program)

-- MODEL

type alias Model =
    String

init : ( Model, Cmd Msg )
init =
    ( "Hello", Cmd.none )

-- MESSAGES

type Msg
    = NoOp

-- VIEW

view : Model -> Html Msg
view model =
    div []
        [ text model ]

-- UPDATE

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        NoOp ->
            ( model, Cmd.none )

-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none

-- MAIN

main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```


This page covers Tutorial v2. Elm 0.18.

Webpack 4

package.json

Finally we want to add some npm scripts so we can run our servers easily. In **package.json** replace `scripts` with:

```
"scripts": {  
  "api": "node api.js",  
  "build": "webpack",  
  "watch": "webpack --watch",  
  "dev": "webpack-dev-server --port 3000"  
},
```

- So now `npm run api` will run our fake server.
- `npm run build` will create a webpack build and put the bundles in `dist`.
- `npm run watch` runs the webpack watcher which puts the bundles in `dist` as we change our source code.
- `npm run dev` runs the webpack dev server.

Node Foreman

We have two servers to run for developing: the Api and the Frontend, we will need to launch both manually to test our application, this is ok but there is a nicer way.

Install Node Foreman:

```
npm install -g foreman
```

Then create a file called `Procfile` in the root of the project with:

```
api: npm run api  
client: npm run dev
```

This will give us a cli command called `nf` that allows to launch and kill both processes at the same time.

Test it

Let's test our setup

In a terminal window run:

```
nf start
```

If you browse to `http://localhost:3000/` you should see our application, which outputs "Hello". Use `ctrl-c` to stop the servers.

Your application code should look like <https://github.com/sporto/elm-tutorial-app/tree/018-02-webpack>.

This page covers Tutorial v2. Elm 0.18.

Multiple modules

Our application is going to grow soon, so keeping things in one file will become hard to maintain quite fast.

Circular dependencies

Another issue we are likely to hit at some point will be circular dependencies. For example we might have:

- A `Main` module which has a `Player` type on it.
- A `View` module that imports the `Player` type declared in `Main`.
- `Main` importing `View` to render the view.

We now have a circular dependency:

```
Main --> View
View --> Main
```

How to break it?

In this case what we need to do is to move the `Player` type out of `Main`, somewhere it can be imported by both `Main` and `View`.

To deal with circular dependencies in Elm the easiest thing to do is to split your application into smaller modules. In this particular example we can create another module that can be imported by both `Main` and `View`. We will have three modules:

- `Main`
- `View`
- `Models` (contains the `Player` type)

Now the dependencies will be:

```
Main --> Models
View --> Models
```

There is no circular dependency anymore.

Try creating separate modules for things like **messages**, **models**, **commands** and **utilities**, which are modules that are usually imported by many components.

Let's break the application in smaller modules:

src/Msgs.elm

```
module Msgs exposing (..)

type Msg
  = NoOp
```

src/Models.elm

```
module Models exposing (..)
```

```
type alias Model =  
    String
```

src/Update.elm

```
module Update exposing (..)
```

```
import Msgs exposing (Msg(..))  
import Models exposing (Model)
```

```
update : Msg -> Model -> ( Model, Cmd Msg )  
update msg model =  
    case msg of  
        NoOp ->  
            ( model, Cmd.none )
```

src/View.elm

```
module View exposing (..)
```

```
import Html exposing (Html, div, text)  
import Msgs exposing (Msg)  
import Models exposing (Model)
```

```
view : Model -> Html Msg  
view model =  
    div []  
        [ text model ]
```

src/Main.elm

```
module Main exposing (..)

import Html exposing (Html, div, text, program)
import Msgs exposing (Msg)
import Models exposing (Model)
import Update exposing (update)
import View exposing (view)

init : ( Model, Cmd Msg )
init =
    ( "Hello", Cmd.none )

subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none

-- MAIN

main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

You can find the code here <https://github.com/sporto/elm-tutorial-app/tree/018-v02-03-multiple-modules>

There are lots of little modules now, this is overkill for a trivial application. But for a bigger application splitting it makes it easier to work with.

This page covers Tutorial v2. Elm 0.18.

Resources

Up to this point your code should look like <https://github.com/sporto/elm-tutorial-app/tree/018-v02-03-multiple-modules>

In this chapter we will add the first resource to our application: Players.

This page covers Tutorial v2. Elm 0.18.

The Players resource

We will organise our application structure by the name of the resources in our application. In this app, we only have one resource (`Players`) so there will be only a `Players` directory.

The `Players` directory will the views for players: A list and a edit view. Each view will have its own Elm module:

- `Players/List.elm`
- `Players/Edit.elm`

This page covers Tutorial v2. Elm 0.18.

Players Model

Change **src/Models.elm** to:

```
module Models exposing (..)

type alias Model =
  { players : List Player
  }

initialModel : Model
initialModel =
  { players = [ Player "1" "Sam" 1 ]
  }

type alias PlayerId =
  String

type alias Player =
  { id : PlayerId
  , name : String
  , level : Int
  }
```

Here we define how a player record looks. It has an id, a name and a level.

Also note the definition for `PlayerId`, it is just an alias to `String`, doing this is useful for clarity later on when we have function that takes many ids. For example:

```
addPerkToPlayer : Int -> String -> Player
```

is much clearer when written as:

```
addPerkToPlayer : PerkId -> PlayerId -> Player
```

We also added `players` to our main model and created a hardcoded list for now.

This page covers Tutorial v2. Elm 0.18.

Main

We want to use the `initialModel` we created before. Modify **src/Main.elm** to use this:

```
module Main exposing (..)

import Html exposing (program)
import Msgs exposing (Msg)
import Models exposing (Model, initialModel)
import Update exposing (update)
import View exposing (view)

init : ( Model, Cmd Msg )
init =
    ( initialModel, Cmd.none )

subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none

-- MAIN

main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

Here we added `initialModel` in the import and `init` .

This page covers Tutorial v2. Elm 0.18.

Players List

Create **src/Players/List.elm**

```
module Players.List exposing (..)

import Html exposing (..)
import Html.Attributes exposing (class)
import Msgs exposing (Msg)
import Models exposing (Player)

view : List Player -> Html Msg
view players =
    div []
        [ nav
        , list players
        ]

nav : Html Msg
nav =
    div [ class "clearfix mb2 white bg-black" ]
        [ div [ class "left p2" ] [ text "Players" ] ]

list : List Player -> Html Msg
list players =
    div [ class "p2" ]
        [ table []
            [ thead []
                [ tr []
                    [ th [] [ text "Id" ]
                    , th [] [ text "Name" ]
                    , th [] [ text "Level" ]
                    , th [] [ text "Actions" ]
                    ]
                ]
            , tbody [] (List.map playerRow players)
            ]
        ]

playerRow : Player -> Html Msg
playerRow player =
    tr []
        [ td [] [ text player.id ]
        , td [] [ text player.name ]
        , td [] [ text (toString player.level) ]
        , td []
            []
        ]
```

This view shows a list of players.

This page covers Tutorial v2. Elm 0.18.

Main View

Modify **src/View.elm** to include the list of players:

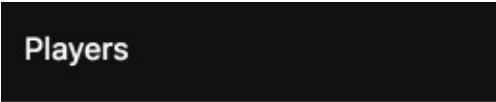
```
module View exposing (..)

import Html exposing (Html, div, text)
import Msgs exposing (Msg)
import Models exposing (Model)
import Players.List

view : Model -> Html Msg
view model =
    div []
        [ page model ]

page : Model -> Html Msg
page model =
    Players.List.view model.players
```

When you run the application you should see a list with one user.



Players

Id Name Level Actions

1 Sam 1

The application should look like <https://github.com/sporto/elm-tutorial-app/tree/018-v02-04-resources>

This page covers Tutorial v2. Elm 0.18.

Fetching resources

This chapter covers fetching the players' collection from the fake API.

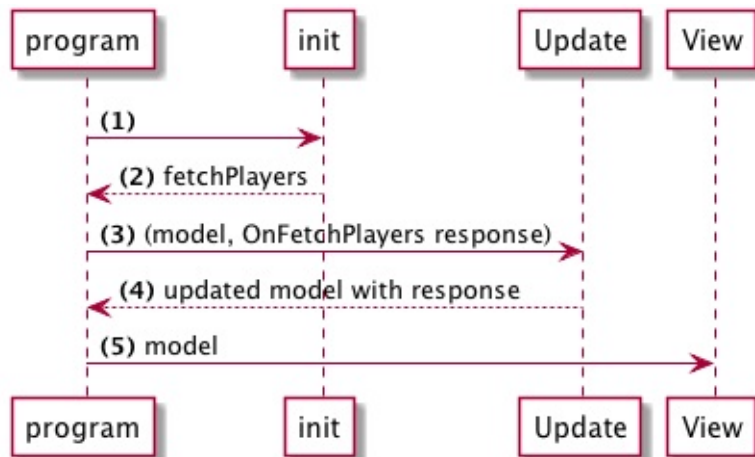
Up to this point your application code looks like <https://github.com/sporto/elm-tutorial-app/tree/018-v02-04-resources>

This page covers Tutorial v2. Elm 0.18.

Plan

The next step is to fetch the list of players from the fake API we created before.

This is the plan:



(1-2) When the application loads, we trigger a command to initiate an Http request to fetch the players. This will be done in the `init` of `Html.program`.

(3-4) When the request is done, we trigger a `OnFetchPlayers` with the response, this message flows to `Update` which updates the model by storing the response.

(5) Then the application renders with the updated players' list.

Dependencies

We will need a few new packages, install them using:

```
elm package install elm-lang/http
elm package install NoRedInk/elm-decode-pipeline
elm package install krisajenkins/remotedata
```

- `elm-lang/http` is used for sending http requests.
- `NoRedInk/elm-decode-pipeline` offers an alternative and cleaner API for decoding JSON.
- `krisajenkins/remotedata` offers a robust pattern for handling remote resources, we will more about this later.

This page covers Tutorial v2. Elm 0.18.

Messages

First let's create the messages we need for fetching players. Add a new import and message to **src/Msgs.elm**

```
module Msgs exposing (..)

import Models exposing (Player)
import RemoteData exposing (WebData)

type Msg
    = OnFetchPlayers (WebData (List Player))
```

`OnFetchPlayers` will be called when we get the response from the server. This message will carry a `WebData (List Player)`.

`WebData` is a type that provides four constructors: `NotAsked`, `Loading`, `Success` and `Failure`. These four possible constructors describe all states in which an HTTP resource could be. Read more about this [here](#).

This page covers Tutorial v2. Elm 0.18.

Models

We want to store the response from the server in the models instead of the hardcoded list of players we have now. In **src/Models.elm**, include a new import and change the type of `players` :

```
...  
  
import RemoteData exposing (WebData)  
  
type alias Model =  
  { players : WebData (List Player)  
  }  
  
initialModel : Model  
initialModel =  
  { players = RemoteData.Loading  
  }  
  
...
```

- We changed the type of `players` from `List Player` to `WebData (List Player)` . This type `WebData` will contain the list of players when we get a successful response from the API.
- Our initial `players` attribute will be `RemoteData.Loading` , as it says this indicates that the resource is loading.

This page covers Tutorial v2. Elm 0.18.

Commands

Now we need a command to fetch the players from the server. Create **src/Commands.elm**:

```
module Commands exposing (..)

import Http
import Json.Decode as Decode
import Json.Decode.Pipeline exposing (decode, required)
import Msgs exposing (Msg)
import Models exposing (PlayerId, Player)
import RemoteData

fetchPlayers : Cmd Msg
fetchPlayers =
    Http.get fetchPlayersUrl playersDecoder
        |> RemoteData.sendRequest
        |> Cmd.map Msgs.OnFetchPlayers

fetchPlayersUrl : String
fetchPlayersUrl =
    "http://localhost:4000/players"

playersDecoder : Decode.Decoder (List Player)
playersDecoder =
    Decode.list playerDecoder

playerDecoder : Decode.Decoder Player
playerDecoder =
    decode Player
        |> required "id" Decode.string
        |> required "name" Decode.string
        |> required "level" Decode.int
```

Let's go through this code.

```
fetchPlayers : Cmd Msg
fetchPlayers =
    Http.get fetchPlayersUrl playersDecoder ❶
        |> RemoteData.sendRequest ❷
        |> Cmd.map Msgs.OnFetchPlayers ❸
```

Here we create a command for our application to run.

- ❶ `Http.get` takes a url and a decoder. This returns a `Request` type.
- ❷ We pass this request to `RemoteData.sendRequest`, this function will wrap the request in a `WebData` type and return a `Cmd` to sends the request.
- ❸ We map the command from `RemoteData` to `OnFetchPlayers`. In that way we can handle the response in our update.

```
playersDecoder : Decode.Decoder (List Player)
playersDecoder =
    Decode.list playerDecoder
```

This decoder delegates the decoding of each member of a list to `playerDecoder`

```
playerDecoder : Decode.Decoder Player
playerDecoder =
  decode Player
    |> required "id" Decode.string
    |> required "name" Decode.string
    |> required "level" Decode.int
```

`playerDecoder` creates a JSON decoder that returns a `Player` record. Here we use `decode` and `required` from the JSON Pipeline package. This package allows to decode JSON in a more intuitive way than doing it without any packages.

Remember that none of this actually executes until we send the command to **program**.

Now that we have a command to fetch players we need to call it.

In **src/Main.elm** call this command in `init` :

```
...
import Commands exposing (fetchPlayers)

init : ( Model, Cmd Msg )
init =
  ( initialModel, fetchPlayers )
```

Here we import `fetchPlayers` and call this in `init` . This tells `Html.program` to run this request when the application starts.

This page covers Tutorial v2. Elm 0.18.

Update

When the request for players is done, we trigger the `onFetchAll` message.

src/Update.elm should account for this new message. Change `update` to:

```
module Update exposing (..)

import Msgs exposing (Msg)
import Models exposing (Model)

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    Msgs.OnFetchPlayers response ->
      ( { model | players = response }, Cmd.none )
```

When we get `onFetchPlayers` we get a response and store it `players` .

This page covers Tutorial v2. Elm 0.18.

Views

After receiving the response back we store it in `players`. We now want to display the list of players.

The attribute `players` was a list of players before (`List Player`), now is `WebData (List Player)` type. Let's change **src/Players/List.elm** to handle the new type of `players`.

Add `RemoteData` import

```
import RemoteData exposing (WebData)
```

Change `view` to:

```
view : WebData (List Player) -> Html Msg
view response =
    div []
        [ nav
          , maybeList response
        ]
```

Here we changed the signature and call a new function `maybeList`. Add `maybeList`:

```
maybeList : WebData (List Player) -> Html Msg
maybeList response =
    case response of
        RemoteData.NotAsked ->
            text ""

        RemoteData.Loading ->
            text "Loading..."

        RemoteData.Success players ->
            list players

        RemoteData.Failure error ->
            text (toString error)
```

This function uses a case expression to pattern match on the type of `response`. This types are provided by the `RemoteData` package.

If `response` is of type `Success` we display the list of players. We call the previous `list` function we already had.

This page covers Tutorial v2. Elm 0.18.

Try it

Try it! Run the app in one terminal using:

```
nf start
```

Refresh the browser, our application should now fetch the list of players from the server. The app should look like:

Players

	Id	Name	Level	Actions
2	Lance	1		
3	Aki	3		
4	Maria	4		
5	Julio	1		
6	Julian	1		
7	Jaime	1		

Your application code should look at this stage like <https://github.com/sporto/elm-tutorial-app/tree/018-v02-05-fetch>.

This page covers Tutorial v2. Elm 0.18.

Routing

This chapter covers adding routing to our application.

Up to this point the application code looks like <https://github.com/sporto/elm-tutorial-app/tree/018-v02-05-fetch>

This page covers Tutorial v2. Elm 0.18.

Routing introduction

Let's add a routing to our application. We will be using the [Elm Navigation package](#) and [UrlParser](#).

- Navigation provides the means to change the browser location and respond to changes
- UrlParser provides route matchers

First install the packages:

```
elm package install elm-lang/navigation
elm package install evancz/url-parser
```

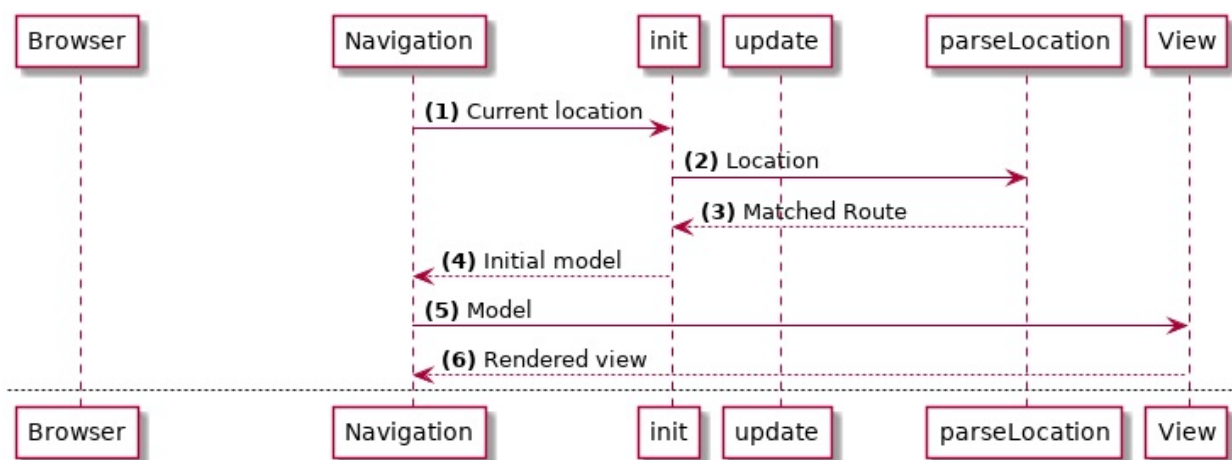
`Navigation` is a library that wraps `Html.program`. It has all the functionality of `Html.program` plus some extra things:

- Listens for location changes on the browser
- Triggers a message when the location changes
- Provides ways of changing the browser location

Flow

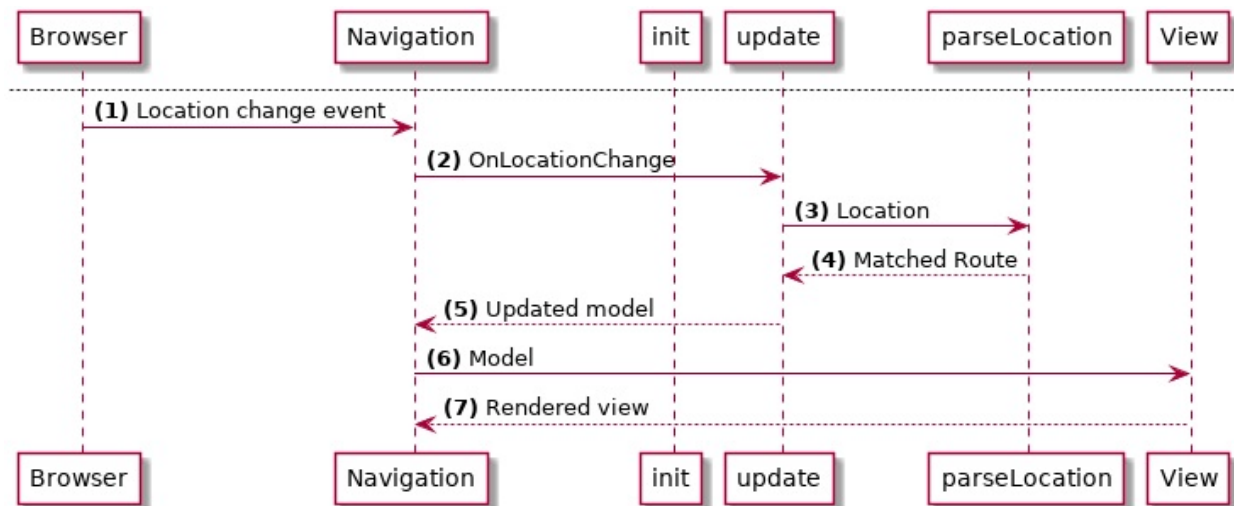
Here are a couple of diagrams to understand how routing will work.

Initial render



- (1) When the page first loads the `Navigation` module will fetch the current `Location` and send it to the application `init` function.
- (2) In `init` we parse this location and get a matching `Route`.
- (3, 4) We then store this matched `Route` in our initial model and return this model to `Navigation`.
- (5, 6) `Navigation` then renders the view by sending this initial model.

When the location changes



- (1) When the browser location changes the Navigation library receives an event
- (2) A `OnLocationChange` message is sent to our `update` function. This message will contain the new `Location`.
- (3, 4) In `update` we parse the `Location` and return the matching `Route`.
- (5) From `update` we return the updated model which contains the update `Route`.
- (6, 7) Navigation then renders the application as normal

This page covers Tutorial v2. Elm 0.18.

Models

In **src/Models.elm** we will define the possible routes for our application. Add a new type:

```
type Route
  = PlayersRoute
  | PlayerRoute PlayerId
  | NotFoundRoute
```

`NotFoundRoute` will be used when no route matches the browser path.

Routing

Create a module **src/Routing.elm** for defining the application routing configuration.

In this module we define:

- how to match browser paths to routes using path matchers
- how to react to routing messages

In **src/Routing.elm**:

```
module Routing exposing (..)

import Navigation exposing (Location)
import Models exposing (PlayerId, Route(..))
import UrlParser exposing (..)

matchers : Parser (Route -> a) a
matchers =
  oneOf
    [ map PlayersRoute top
    , map PlayerRoute (s "players" </> string)
    , map PlayersRoute (s "players")
    ]

parseLocation : Location -> Route
parseLocation location =
  case (parseHash matchers location) of
    Just route ->
      route

    Nothing ->
      NotFoundRoute
```

Let's go over this module.

Matchers

```
matchers : Parser (Route -> a) a
matchers =
  oneOf
    [ map PlayersRoute top
    , map PlayerRoute (s "players" </> string)
    , map PlayersRoute (s "players")
    ]
```

Here we define route matchers. These parsers are provided by the url-parser library.

We want three matchers:

- One for the top route which will resolve to `PlayersRoute`
- One for `/players` which will resolve to `PlayersRoute` as well
- And one for `/players/id` which will resolve to `PlayerRoute id`

Note that the order is important.

See more details about this library here <http://package.elm-lang.org/packages/evancz/url-parser>.

parseLocation

```
parseLocation : Location -> Route
parseLocation location =
  case (parseHash matchers location) of
    Just route ->
      route

    Nothing ->
      NotFoundRoute
```

Each time the browser location changes, the Navigation library will trigger a message containing a `Navigation.Location` record. From our main `update` we will call `parseLocation` with this record.

`parseLocation` is a function that parses this `Location` record and returns the matched `Route` if possible. If all matchers fail we return `NotFoundRoute`.

In this case we use `UrlParser.parseHash` as we will be routing using the hash. You could use `UrlParser.parsePath` to route with the path instead.

This page covers Tutorial v2. Elm 0.18.

Player edit view

We need a new view to show when hitting `/players/3`.

Create **src/Players/Edit.elm**:

```
module Players.Edit exposing (..)

import Html exposing (..)
import Html.Attributes exposing (class, value, href)
import Msgs exposing (Msg)
import Models exposing (Player)

view : Player -> Html Msg
view model =
    div []
        [ nav model
        , form model
        ]

nav : Player -> Html Msg
nav model =
    div [ class "clearfix mb2 white bg-black p1" ]
        []

form : Player -> Html Msg
form player =
    div [ class "m3" ]
        [ h1 [] [ text player.name ]
        , formLevel player
        ]

formLevel : Player -> Html Msg
formLevel player =
    div
        [ class "clearfix py1"
        ]
        [ div [ class "col col-5" ] [ text "Level" ]
        , div [ class "col col-7" ]
            [ span [ class "h2 bold" ] [ text (toString player.level) ]
            , btnLevelDecrease player
            , btnLevelIncrease player
            ]
        ]

btnLevelDecrease : Player -> Html Msg
btnLevelDecrease player =
    a [ class "btn m1 h1" ]
        [ i [ class "fa fa-minus-circle" ] [] ]

btnLevelIncrease : Player -> Html Msg
btnLevelIncrease player =
    a [ class "btn m1 h1" ]
        [ i [ class "fa fa-plus-circle" ] [] ]
```

This view shows a form with the player's level. At the moment we have some dummy buttons that will be implemented later e.g. `btnLevelIncrease`.

This page covers Tutorial v2. Elm 0.18.

Main model

In our main application model we want to store the current route. In **src/Models.elm**, change `Model` and `initialModel` to:

```
...

type alias Model =
  { players : WebData (List Player)
  , route : Route
  }

initialModel : Route -> Model
initialModel route =
  { players = RemoteData.Loading
  , route = route
  }
```

Here we:

- added `route` to the model
- changed `initialModel` so it takes a `route`

This page covers Tutorial v2. Elm 0.18.

Messages

When the browser location changes we will trigger a new `OnLocationChange` message.

Change **src/Messages.elm** to:

```
module Msgs exposing (..)

import Models exposing (Player)
import Navigation exposing (Location)
import RemoteData exposing (WebData)

type Msg
  = OnFetchPlayers (WebData (List Player))
  | OnLocationChange Location
```

- We are now importing `Navigation`
- And we added a `OnLocationChange Location` message

This page covers Tutorial v2. Elm 0.18.

Update

We need our `update` function to respond to the new `OnLocationChange` message.

In **src/Update.elm** add a new branch:

```
...
import Routing exposing (parseLocation)

...

update msg model =
  case msg of
    ...
    Msgs.OnLocationChange location ->
      let
        newRoute =
          parseLocation location
      in
        ( { model | route = newRoute }, Cmd.none )
```

Here when we receive the `OnLocationChange` message, we parse this location and store the matched route in the model.

This page covers Tutorial v2. Elm 0.18.

Main view

Our main application view needs to show different pages as we change the browser location.

Change **src/View.elm** to:

```
module View exposing (..)

import Html exposing (Html, div, text)
import Models exposing (Model, PlayerId)
import Models exposing (Model)
import Msgs exposing (Msg)
import Players.Edit
import Players.List
import RemoteData

view : Model -> Html Msg
view model =
    div []
        [ page model ]

page : Model -> Html Msg
page model =
    case model.route of
        Models.PlayersRoute ->
            Players.List.view model.players

        Models.PlayerRoute id ->
            playerEditPage model id

        Models.NotFoundRoute ->
            notFoundView

playerEditPage : Model -> PlayerId -> Html Msg
playerEditPage model playerId =
    case model.players of
        RemoteData.NotAsked ->
            text ""

        RemoteData.Loading ->
            text "Loading ..."

        RemoteData.Success players ->
            let
                maybePlayer =
                    players
                    |> List.filter (\player -> player.id == playerId)
                    |> List.head
            in
            case maybePlayer of
                Just player ->
                    Players.Edit.view player

                Nothing ->
                    notFoundView

        RemoteData.Failure err ->
            text (toString err)

notFoundView : Html msg
notFoundView =
    div []
        [ text "Not found"
        ]
```

Showing the correct view

```

page : Model -> Html Msg
page model =
  case model.route of
    Models.PlayersRoute ->
      Players.List.view model.players

    Models.PlayerRoute id ->
      playerEditPage model id

    Models.NotFoundRoute ->
      notFoundView

```

Now we have a function `page` which has a case expression to show the correct view depending on what is in `model.route`.

When the player edit route matches (e.g. `#players/2`) we will get the player id from the route i.e. `PlayerRoute playerId`.

The player edit page

```

playerEditPage : Model -> PlayerId -> Html Msg
playerEditPage model playerId =
  case model.players of ❶
    RemoteData.NotAsked ->
      text ""

    RemoteData.Loading ->
      text "Loading ..."

    RemoteData.Success players ->
      let
        maybePlayer =
          players
            |> List.filter (\player -> player.id == playerId)
            |> List.head
      in
        case maybePlayer of ❷
          Just player ->
            Players.Edit.view player

          Nothing ->
            notFoundView

    RemoteData.Failure err ->
      text (toString err)

```

When navigate to a page we have the `playerId`, but we might not have the list of players loaded or the actual player record for that id in the list. We need to account for this two cases.

- ❶ So first we check the type of `model.players`, we will only show the list if this attribute is a `RemoteData.Success list`.
- ❷ Then we filter the players' list by that id and have a case expression that show the correct view depending if the player is found or not.

notFoundView

`notFoundView` is shown when no route matches. Note the type `Html msg` instead of `Html Msg`. This is because this view doesn't produce any messages so can use a generic type variable `msg` instead of a specific type `Msg`.

This page covers Tutorial v2. Elm 0.18.

Main

Finally we need to wire everything in the Main module.

Change **src/Main.elm** to:

```
module Main exposing (..)

import Commands exposing (fetchPlayers)
import Models exposing (Model, initialModel)
import Msgs exposing (Msg)
import Navigation exposing (Location)
import Routing
import Update exposing (update)
import View exposing (view)

init : Location -> ( Model, Cmd Msg )
init location =
    let
        currentRoute =
            Routing.parseLocation location
    in
        ( initialModel currentRoute, fetchPlayers )

subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none

main : Program Never Model Msg
main =
    Navigation.program Msgs.OnLocationChange
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

New imports

We added imports for `Navigation` and `Routing` .

Init

```
init : Location -> ( Model, Cmd Msg )
init location =
    let
        currentRoute =
            Routing.parseLocation location
    in
        ( initialModel currentRoute, fetchPlayers )
```

Our init function will now take an initial `Location` from the `Navigation` package. We parse this `Location` using the `parseLocation` function we created before. Then we store this initial **route** in our model.

main

`main` now uses `Navigation.program` instead of `Html.program`. `Navigation.program` wraps `Html.program` but also triggers a message when the browser location changes. In our case this message will be `OnLocationChange`.

This page covers Tutorial v2. Elm 0.18.

Try it

Let's try what we have so far. Run the application by doing:

```
nf start
```

Then go to `http://localhost:3000` in your browser. You should see the list of users.

Players

	Id	Name	Level	Actions
2	Lance	1		
3	Aki	3		
4	Maria	4		
5	Julio	1		
6	Julian	1		
7	Jaime	1		

If you go to `http://localhost:3000/#players/2` then you should see one user.

Lance

Level

1



Next we will add some navigation.

Up to this point your application code should look <https://github.com/sporto/elm-tutorial-app/tree/018-v02-06-routing>.

This page covers Tutorial v2. Elm 0.18.

Navigation

Next let's add buttons to navigate between views.

Routing

In **src/Routing.elm** add two new functions:

```
playersPath : String
playersPath =
    "#players"

playerPath : PlayerId -> String
playerPath id =
    "#players/" ++ id
```

Players List

The players' list needs to show a button for each player to trigger the `ShowPlayer` message.

In **src/Players/List.elm**. First import `href` and `Routing` :

```
import Html.Attributes exposing (class, href)
...
import Routing exposing (playerPath)
```

Add a new function for this button at the end:

```
editBtn : Player -> Html.Html Msg
editBtn player =
    let
        path =
            playerPath player.id
    in
        a
            [ class "btn regular"
            , href path
            ]
            [ i [ class "fa fa-pencil mr1" ] [], text "Edit" ]
```

This button is a common `a` tag, which will change the browser url directly. As we are using hash routing we can just change the location hash and routing will work.

And change `playerRow` to include this button:

```
playerRow : Player -> Html.Html Msg
playerRow player =
    tr []
        [ td [] [ text (toString player.id) ]
        , td [] [ text player.name ]
        , td [] [ text (toString player.level) ]
        , td []
            [ editBtn player ]
        ]
```

Player Edit

Let's add the navigation button to the edit view. In `/src/Players/Edit.elm`:

Add the imports:

```
import Html.Attributes exposing (class, value, href)
import Routing exposing (playersPath)
```

Add a new function at the end for the list button:

```
listBtn : Html Msg
listBtn =
  a
    [ class "btn regular"
    , href playersPath
    ]
    [ i [ class "fa fa-chevron-left mr1" ] [], text "List" ]
```

And add this button to the list, change the `nav` function to:

```
nav : Player -> Html Msg
nav model =
  div [ class "clearfix mb2 white bg-black p1" ]
    [ listBtn ]
```

This page covers Tutorial v2. Elm 0.18.

Test it

Go to the list `http://localhost:3000/#players` . You should now see an Edit button, upon clicking it the location should change to the edit view.

Up to this point your application code should look <https://github.com/sporto/elm-tutorial-app/tree/018-v02-07-navigation>.

Navigation approaches

We are using hash routing here which is simple. But it is a bit ugly because your URLs will need to include a `#` . Hash routing will also conflict if you want to use the `#` for its original intent, which is creating anchors on a page.

As an alternative you can use "path" routing, which uses push state. Instead of having something like `app.com/#users` you will have `app.com/users` , which is nicer.

You can use path routing in Elm using the `Navigation` module too. See [this repository for an explanation and example](#).

This page covers Tutorial v2. Elm 0.18.

Edit

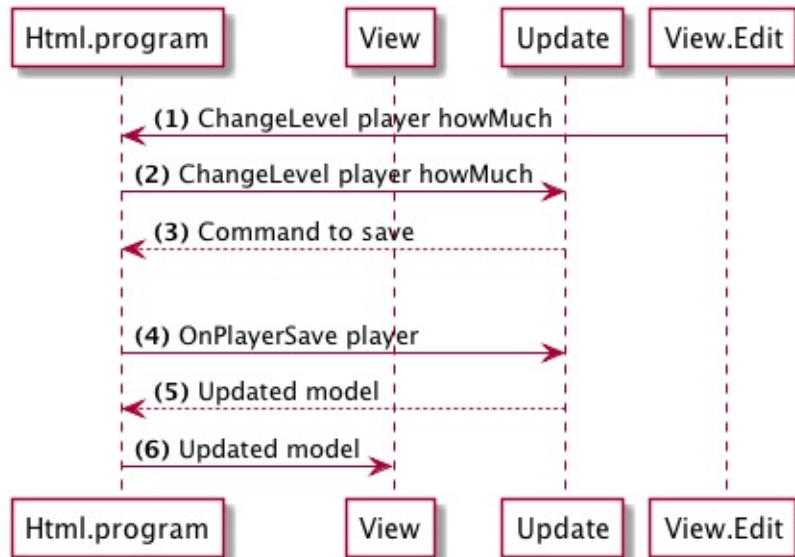
In this last chapter of the tutorial we will edit a player's level and save it to the backend.

Up to this point your application code should look like <https://github.com/sporto/elm-tutorial-app/tree/018-v02-07-navigation>.

This page covers Tutorial v2. Elm 0.18.

Plan

The plan for changing a player's level is as follows:



(1) When the user clicks the increase or decrease button we trigger a message `ChangeLevel` with the `player` and `howMuch` as payload.

(2) **Html.program** (which Navigation wraps) will send this message back to `Update`.

(3) `Update` will return a command to save the player.

(4) The Elm runtime executes the command (trigger an API call) and we will get a result back, which is either a successful save or a failure. In the success case we trigger a `OnPlayerSave` message with the updated player as payload.

(5) In `Update` we update the `players` model and return it.

(6) Then **Html.program** will render the application with the updated model.

This page covers Tutorial v2. Elm 0.18.

Messages

Let's start by adding the messages we will need.

In **src/Msgs.elm** add:

```
import Http
...

type Msg
  ...
  | ChangeLevel Player Int
  | OnPlayerSave (Result Http.Error Player)
```

- `ChangeLevel` will trigger when the user wants to change the level. The second parameter is an integer that indicates how much to change the level e.g. -1 to decrease or 1 to increase.
- Then we will send a request to update the player to the API. `OnPlayerSave` will be triggered after the response from the API is received.
- `OnPlayerSave` will either carry the updated player on success or the Http error on failure.

This page covers Tutorial v2. Elm 0.18.

Player edit view

We created a `ChangeLevel` message. Let's trigger this message from the player's edit view.

In `src/Players/Edit.elm`, first add `onClick` :

```
import Html.Events exposing (onClick)
```

And change `btnLevelDecrease` and `btnLevelIncrease` :

```
...
btnLevelDecrease : Player -> Html Msg
btnLevelDecrease player =
    let
        message =
            Msgs.ChangeLevel player -1
    in
        a [ class "btn ml1 h1", onClick message ]
          [ i [ class "fa fa-minus-circle" ] [] ]

btnLevelIncrease : Player -> Html Msg
btnLevelIncrease player =
    let
        message =
            Msgs.ChangeLevel player 1
    in
        a [ class "btn ml1 h1", onClick message ]
          [ i [ class "fa fa-plus-circle" ] [] ]
```

In these two buttons we added `onClick message` . This message is `Msgs.ChangeLevel player howMuch` . Where `howMuch` is `-1` to decrease level and `1` to increase it.

This page covers Tutorial v2. Elm 0.18.

Players Commands

Next let's create the command to updated the player through our API.

In `src/Players/Commands.elm` add:

```
import Json.Encode as Encode

...

savePlayerUrl : PlayerId -> String
savePlayerUrl playerId =
    "http://localhost:4000/players/" ++ playerId

savePlayerRequest : Player -> Http.Request Player
savePlayerRequest player =
    Http.request
        { body = playerEncoder player |> Http.jsonBody
        , expect = Http.expectJson playerDecoder
        , headers = []
        , method = "PATCH"
        , timeout = Nothing
        , url = savePlayerUrl player.id
        , withCredentials = False
        }

savePlayerCmd : Player -> Cmd Msg
savePlayerCmd player =
    savePlayerRequest player
    |> Http.send Msgs.OnPlayerSave

playerEncoder : Player -> Encode.Value
playerEncoder player =
    let
        attributes =
            [ ( "id", Encode.string player.id )
            , ( "name", Encode.string player.name )
            , ( "level", Encode.int player.level )
            ]
    in
        Encode.object attributes
```

Save request

```
savePlayerRequest : Player -> Http.Request Player
savePlayerRequest player =
    Http.request
        { body = playerEncoder player |> Http.jsonBody ❶
        , expect = Http.expectJson playerDecoder ❷
        , headers = []
        , method = "PATCH" ❸
        , timeout = Nothing
        , url = savePlayerUrl player.id
        , withCredentials = False
        }
```

❶ Here we encode the given player and then convert the encoded value to a JSON string ❷ Here we specify how to parse the response, in this case we want to parse the returned JSON back into an Elm value. ❸ `PATCH` is the http method that our API expects when updating records.

Save

```
savePlayerCmd : Player -> Cmd Msg
savePlayerCmd player =
  savePlayerRequest player ❶
    |> Http.send Msgs.OnPlayerSave ❷
```

Here we create the save request ❶ and then generate a command to send the request using `Http.send` ❷. `Http.send` takes a message constructor (`OnPlayerSave` in this case). After the request is done, Elm will trigger the `OnPlayerSave` message with the response for the request.

This page covers Tutorial v2. Elm 0.18.

Update

Finally we need to account for the new messages in our `update` function. In `src/Update.elm`:

Add a new imports:

```
import Commands exposing (savePlayerCmd)
import Models exposing (Model, Player)
import RemoteData
```

New messages

Update branches to update to handle the newly created messages:

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    ...

    Msgs.ChangeLevel player howMuch ->
      let
        updatedPlayer =
          { player | level = player.level + howMuch }
      in
        ( model, savePlayerCmd updatedPlayer )

    Msgs.OnPlayerSave (Ok player) ->
      ( updatePlayer model player, Cmd.none )

    Msgs.OnPlayerSave (Err error) ->
      ( model, Cmd.none )
```

ChangeLevel

In `ChangeLevel` we first update the `level` attribute for the given player and then return a command `savePlayerCmd` to save it.

OnPlayerSave

When we get `OnPlayerSave` back we pattern match so we handle the success and failure case differently. On the failure case we are just discarding the error and leaving the model as it was. This is not great but for simplicity we will do it like this.

In the success case we are calling a helper function `updatePlayer` to update the changed player, we will write this function next.

Update the player

Add a helper function to update the player:

```
updatePlayer : Model -> Player -> Model
updatePlayer model updatedPlayer =
  let
    pick currentPlayer =
      if updatedPlayer.id == currentPlayer.id then
        updatedPlayer
      else
        currentPlayer

    updatePlayerList players =
      List.map pick players

    updatedPlayers =
      RemoteData.map updatePlayerList model.players
  in
    { model | players = updatedPlayers }
```

This function is called after we get an updated player from the API. This function needs to swap an existing player in our model for the updated player coming from the API.

We don't know what is in `model.players`, it could be `RemoteData.Loading` or `RemoteData.Success players` or some other case. So first we need to account for this. `RemoteData` provides a `map` function that only applies when we have `RemoteData.Success`. We use this in `updatedPlayers`.

`updatePlayerList` will only be called if `model.players` is `RemoteData.Success players`. `updatePlayerList` is a function that maps over a list of players and swaps the updated player.

Try it

This is all the setup necessary for changing a player's level. Try it, go to the edit view and click the - and + buttons. You should see the level changing and if you refresh your browser that change should be persisted on the server.

Up to this point your application code should look <https://github.com/sporto/elm-tutorial-app/tree/018-v02-08-edit>.

Conclusion

This concludes this tutorial but keep reading for some ideas on improvements and features.

I would like to hear your feedback to improve this tutorial. Please open issues at <https://github.com/sporto/elm-tutorial>.

Thanks!

Improvements

Here is a list of possible improvement you can try on this app.

Create and delete players

I have left this off in order to keep the tutorial short, definitely an important feature.

Change the name of a player

Show an error message when an Http request fails

At the moment if fetching or saving players fail we do nothing. It would be nice to show an error message to the user.

Even better error messages

Even better than just showing error messages it would be great to:

- Show different types of flash messages e.g. error and info
- Show several flash messages at the same time
- Have the ability to dismiss a message
- Remove a message automatically after a few seconds

Optimistic updates

At the moment all update functions are pessimistic. Meaning that they don't change the models until there is a succesful response from the server. One big improvement to the application would be to add optimistic creation, update and deletion. But this will also mean better error handling.

Validations

We should avoid having players without name. One nice feature would be to have a validation on the player's name so it cannot be empty.

Add perks and bonuses

We can add a list of perks that a player can have. These perk would be equipment, apparel, scrolls, accessories, etc. e.g. "Steel sword" would be one. Then we would have associations between players and perks.

Each perk would have a bonus associated with it. Then players will have a calculated strength that is their level plus all the bonuses they have.

For a more featured version of this application see the master branch of <https://github.com/sporto/elm-tutorial-app>.

Contexts

Typical `update` or `view` functions look like:

```
view : Model -> Html Msg
view model =
  ...
```

Or

```
update : Msg -> Model -> (Model, Cmd Msg)
update message model =
  ...
```

It is very easy to get stuck in thinking that you need to pass only the `Model` that belongs to this component. Sometimes you need extra information and is perfectly fine to ask for it. For example:

```
type alias Context =
  { model : Model
  , time : Time
  }

view : Context -> Html Msg
view context =
  ...
```

This function asks for the component model plus a `time` which is defined in its parent's model.

Point free style

Point free is a style of writing a function where you omit one or more arguments in the body. To understand this let's see an example.

Here we have a function that adds 10 to a number:

```
add10 : Int -> Int
add10 x =
    10 + x
```

We can rewrite this using the `+` using a prefix notation:

```
add10 : Int -> Int
add10 x =
    (+) 10 x
```

The argument `x` in this case is not strictly necessary, we could rewrite this as:

```
add10 : Int -> Int
add10 =
    (+) 10
```

Note how `x` is missing as both an input argument to `add10` and argument to `+`. `add10` is still a function that requires an integer to calculate a result. Omitting arguments like this is called **point free style**.

Some more examples

```
select : Int -> List Int -> List Int
select num list =
    List.filter (\x -> x < num) list

select 4 [1, 2, 3, 4, 5] == [1, 2, 3]
```

is the same as:

```
select : Int -> List Int -> List Int
select num =
    List.filter (\x -> x < num)

select 4 [1, 2, 3, 4, 5] == [1, 2, 3]
```

```
process : List Int -> List Int
process list =
    reverse list |> drop 3
```

is the same as:

```
process : List Int -> List Int
process =
    reverse >> drop 3
```


This page covers Elm 0.18

Composing

The following pages explain a composition pattern in Elm where you split your messages, models and views in logical groups.

It looks like:

```
Root
- Model
- Messages
- Update
- Views

Group
- Model
- Messages
- Update
- Views
```

`Group` in this case could be a resource (e.g. users), a UI element (e.g. a dropdown), a sub application.

Use with care

In general it is recommended that you keep your *models* and *messages* shallow. This allows to have less boilerplate when working with Elm. Don't think in term of "components" in Elm. It is better to think about views on one side and messages and models on another, not necessarily coupled.

Composition with the Elm Architecture

Sometimes we still want to compose like this. To understand how this works, let's build an example:

- We will have a parent component `App`
- And a child component `Widget`

Child component

Let's begin with the child component. This is the code for **Widget.elm**.

```
module Widget exposing (..)

import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

-- MODEL

type alias Model =
    { count : Int
    }

initialModel : Model
initialModel =
    { count = 0
    }

-- MESSAGES

type Msg
    = Increase

-- VIEW

view : Model -> Html Msg
view model =
    div []
        [ div [] [ text (toString model.count) ]
        , button [ onClick Increase ] [ text "Click" ]
        ]

-- UPDATE

update : Msg -> Model -> ( Model, Cmd Msg )
update message model =
    case message of
        Increase ->
            ( { model | count = model.count + 1 }, Cmd.none )
```

This component is nearly identical to the application that we made in the last section, except for subscriptions and main.

This component:

- Defines its own messages (Msg)
- Defines its own model
- Provides an `update` function that responds to its own messages, e.g. `Increase`.

Note how the component only knows about things declared here. Both `view` and `update` only use types declared within the component (`Msg` and `Model`).

In the next section we will create the parent component.

This page covers Elm 0.18

Composing

The parent component

This is the code for the parent component.

```
module Main exposing (..)

import Html exposing (Html, program)
import Widget

-- MODEL

type alias AppModel =
    { widgetModel : Widget.Model
    }

initialModel : AppModel
initialModel =
    { widgetModel = Widget.initialModel
    }

init : ( AppModel, Cmd Msg )
init =
    ( initialModel, Cmd.none )

-- MESSAGES

type Msg
    = WidgetMsg Widget.Msg

-- VIEW

view : AppModel -> Html Msg
view model =
    Html.div []
        [ Html.map WidgetMsg (Widget.view model.widgetModel)
        ]

-- UPDATE

update : Msg -> AppModel -> ( AppModel, Cmd Msg )
update message model =
    case message of
        WidgetMsg subMsg ->
            let
                ( updatedWidgetModel, widgetCmd ) =
                    Widget.update subMsg model.widgetModel
            in
                ( { model | widgetModel = updatedWidgetModel }, Cmd.map WidgetMsg widgetCmd )
```

```
-- SUBSCRIPTIONS

subscriptions : AppModel -> Sub Msg
subscriptions model =
  Sub.none

-- APP

main : Program Never AppModel Msg
main =
  program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

Let's review the important sections of this code.

Model

```
type alias AppModel =
{ widgetModel : Widget.Model ❶
}
```

The parent component has its own model. One of the attributes on this model contains the `Widget.Model` ❶. Note how this parent component doesn't need to know about what `Widget.Model` is.

```
initialModel : AppModel
initialModel =
{ widgetModel = Widget.initialModel ❷
}
```

When creating the initial application model, we simply call `Widget.initialModel` ❷ from here.

If you were to have multiple child components, you would do the same for each, for example:

```
initialModel : AppModel
initialModel =
{ navModel = Nav.initialModel,
  , sidebarModel = Sidebar.initialModel,
  , widgetModel = Widget.initialModel
}
```

Or we could have multiple child components of the same type:

```
initialModel : AppModel
initialModel =
{ widgetModels = [Widget.initialModel]
}
```

Messages


```
type Msg
  = WidgetMsg Widget.Msg
```

We use a **union type** that wraps `Widget.Msg` to indicate that a message belongs to that component. This allows our application to route messages to the relevant components (This will become clearer after looking at the update function).

In an application with multiple child components we could have something like:

```
type Msg
  = NavMsg Nav.Msg
  | SidebarMsg Sidebar.Msg
  | WidgetMsg Widget.Msg
```

View

```
view : AppModel -> Html Msg
view model =
  Html.div []
    [ Html.map ❶ WidgetMsg ❷ (Widget.view ❸ model.widgetModel ❹)
    ]
```

The main application `view` renders the `Widget.view` ❸. But `Widget.view` emits `Widget.Msg` so it is incompatible with this view which emits `Main.Msg`.

- We use `Html.map` ❶ to map emitted messages from `Widget.view` to the type we expect (`Msg`). `Html.map` tags messages coming from the sub view using the `WidgetMsg` ❷ tag.
- We only pass the part of the model that the child component cares about i.e. `model.widgetModel` ❹.

Update

```
update : Msg -> AppModel -> (AppModel, Cmd Msg)
update message model =
  case message of
    WidgetMsg ❶ subMsg ❷ ->
      let
        (updatedWidgetModel, widgetCmd) ❸ =
          Widget.update ❹ subMsg model.widgetModel
      in
        ({ model | widgetModel = updatedWidgetModel }, Cmd.map ❺ WidgetMsg widgetCmd)
```

When a `WidgetMsg` ❶ is received by `update` we delegate the update to the child component. But the child component will only update what it cares about, which is the `widgetModel` attribute.

We use pattern matching to extract the `subMsg` ❷ from `WidgetMsg`. This `subMsg` will be the type that `Widget.update` expects.

Using this `subMsg` and `model.widgetModel` we call `Widget.update` ❸. This will return a tuple with an updated `widgetModel` and a command.

We use pattern matching again to destructure ❹ the response from `Widget.update`.

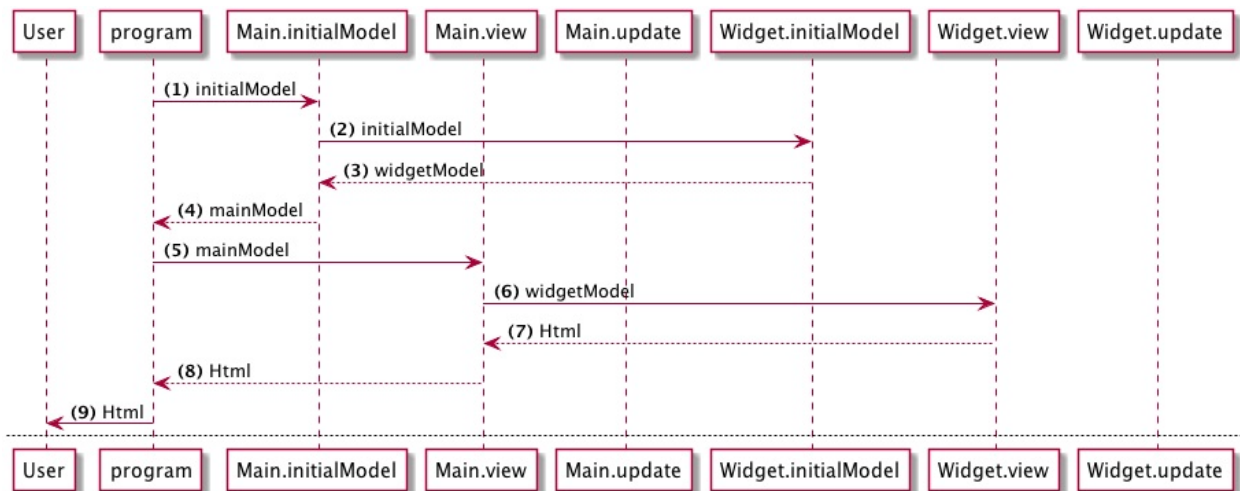
Finally we need to map the command returned by `Widget.update` to the right type. We use `Cmd.map` ❺ for this and tag the command with `WidgetMsg`, similar to what we did in the view.

This page covers Elm 0.18

Composing

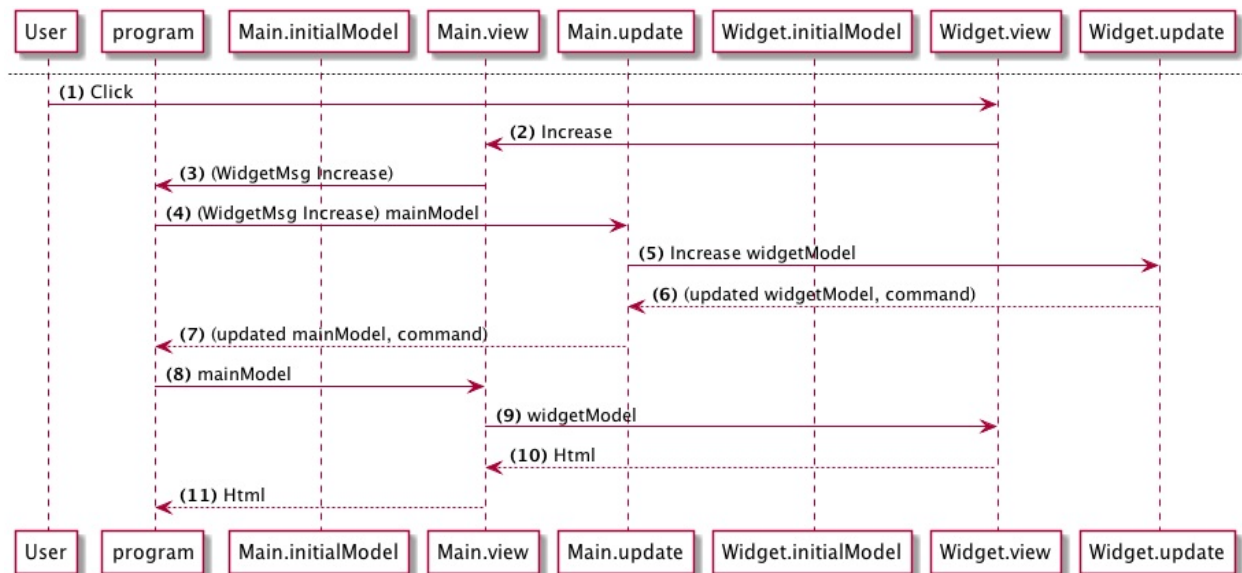
Here are two diagrams that illustrate this architecture:

Initial render



- (1) **program** calls **Main.initialModel** to get the initial model for the application
- (2) **Main** calls **Widget.initialModel**
- (3) **Widget** returns its initial model
- (4) **Main** returns a composed main model which includes the widget model
- (5) **program** calls **Main.view**, passing the **main model**
- (6) **Main.view** calls **Widget.view**, passing the **widgetModel** from the main model
- (7) **Widget.view** returns the rendered Html to **Main**
- (8) **Main.view** returns the rendered Html to **program**
- (9) **program** renders this to the browser.

User interaction



(1) User clicks on the increase button

(2) **Widget.view** emits an **Increase** message which is picked up by **Main.view**.

(3) **Main.view** tags this message so it becomes (WidgetMsg Increase) and it is sent along to **program**

(4) **program** calls **Main.update** with this message and the main model

(5) As the message was tagged with **WidgetMsg**, **Main.update** delegates the update to **Widget.update**, sending along the way the **widgetModel** part of the main model

(6) **Widget.update** modifies the model according to the given message, in this case **Increase**. And returns the modified **widgetModel** plus a command

(7) **Main.update** updates the main model and returns it to **program**

(8) **program** then renders the view again passing the updated main model

Again, don't reach for this initially as a way to organise your application as "components". If you do this too soon you will end up with plenty of boilerplate.

Troubleshooting

If you find some weird compiler behaviour during development try deleting `elm-stuff/build-artifacts` and compiling again, this usually fixes several issues.