

The background of the entire cover is a blue-tinted photograph of a stone tunnel. The tunnel is constructed from rough-hewn stones and has a semi-circular archway at the far end. A bright, white light emanates from the opening of the tunnel, creating a strong contrast with the dark, shadowed interior. The perspective is from inside the tunnel, looking towards the light.

# elm-ui

## THE CSS ESCAPE PLAN

Alex S. Korban



# **elm-ui: The CSS Escape Plan**

Alex S. Korban

© 2020 - 2021 Alex S. Korban

# Contents

<b>1. Introduction</b>	<b>1</b>
1.1 Is it a framework?	2
1.2 How this guide is structured	2
<b>2. Layouts</b>	<b>4</b>
2.1 Getting started: title page layout	4
2.1.1 The menu bar	8
2.2 Going further: a chat page layout	8
2.2.1 Channel and chat panels	10
2.2.2 Filling in the channel list	12
2.2.3 Header and footer sections in the channel	13
2.2.4 Messages	15
2.3 Layout functions	17
2.3.1 Basic layout	18
2.3.2 Padding and spacing	18
2.3.3 Sizing	20
2.3.4 Transparency	21
2.3.5 Alignment	22
2.3.6 Dealing with content overflow	25
2.3.7 Placing elements near, above, or below other elements	26
2.4 Translation, rotation and scaling	29
2.5 Responsive layouts	32
2.6 Debugging	33
2.7 Escape hatches	34

## CONTENTS

<b>3. Page elements and controls</b>	<b>36</b>
3.1 Links	37
3.2 Images	38
3.3 Tables	42
3.4 Buttons	44
3.5 Labels for inputs	46
3.6 Checkboxes	47
3.7 Radio buttons	50
3.8 Single line text inputs	52
3.9 Multiline text inputs	53
3.10 Sliders	54
3.11 A note on focus	57
3.12 Events	57
3.13 Accessibility	58
3.14 Third-party elements and controls	59
<b>4. Typography</b>	<b>60</b>
4.1 Text appearance	60
4.1.1 Font size and typeface	60
4.1.2 Font color, weight and styles	61
4.1.3 Variants	63
4.1.4 Other attributes	64
4.2 Text layout	65
4.2.1 Alignment	71
<b>5. More on styling</b>	<b>72</b>
5.1 Specifying colours	72
5.2 Multiple layouts on a page	73
5.3 Temporary styles	75
<b>6. Working with various types of content</b>	<b>80</b>
6.1 Performance optimisation	80

## CONTENTS

6.2	Forms with composable-form package . . . . .	84
6.3	Markdown . . . . .	95
6.4	Elm-markup . . . . .	100
6.5	Iframes . . . . .	112
6.6	Video . . . . .	113
6.7	Custom elements . . . . .	116
6.8	Drag and drop . . . . .	117
<b>7.</b>	<b>Organising UI code . . . . .</b>	<b>124</b>
7.1	Structuring the code for reuse . . . . .	124
7.2	Stateful UIs . . . . .	131

# 1. Introduction

Several decades in, layout and styling with HTML and CSS remains a complex task.

How many times did you try doing something totally reasonable that *should* be simple with HTML and CSS, only to find yourself getting derailed in bizarre ways? Maybe the text just won't align vertically, or you just can't seem to get the width of the elements right, or the style you've added doesn't seem to have any effect at all.

Unfortunately, this still seems to be an all-too-common experience, which is why I'm excited about the `mdgriffith/elm-ui` package. Its goal is to allow you to build UIs in pure Elm, with HTML and CSS generated for you behind the scenes.

The approach taken by `elm-ui` is based on five ideas:

- Compile-time verification: getting the compiler to verify as much of the layout and styling as possible by defining them in Elm code.
- Composition, reuse and refactoring: allowing the UI code be written and evolved just like the rest of your Elm application.
- Cohesiveness: A simplified approach to layout and styles enabled by a cohesive set of primitives.
- Locality: styles are specified locally for each element.
- Context independence: elements and their attributes are expected to behave the same regardless of the surrounding context (which is often not the case for CSS).

In `elm-ui`, all of the layout and visual styling is done within your `view` function, with the “gross morphology” of layout made explicit through the functions exposed by this package.

I like to say that `elm-ui` has the same relation to HTML and CSS as Elm does to JavaScript. Just like Elm compiles to JavaScript while being a completely different language, `elm-ui`

uses HTML and CSS as its building material, exposing a different paradigm for building UIs to the developer.

While `elm-ui` has its limitations in comparison to HTML and CSS, it nevertheless allows you to create a wide variety of UIs, and also provides some escape hatches such as being able to incorporate `Html` values and HTML attributes.

## 1.1 Is it a framework?

`elm-ui` is, in some sense, a UI framework, however it is fairly low level. It doesn't provide complex widgets such as an autocomplete dropdown or a date picker. It includes a small set of input controls closely aligned with the standard input controls supported by HTML.

A few packages building on top of `elm-ui` have emerged, such as `lucamug/style-framework` and `fabhof/elm-ui-datepicker`, however this is still largely uncharted territory. At present, you should expect to build a lot of interaction from low-level building blocks.

## 1.2 How this guide is structured

I'm assuming that you're already familiar with Elm as well as HTML, CSS and JavaScript. These topics are not covered in the guide.

Due to the nature of the subject, there isn't going to be single example running through the guide as I would have to invent a really contrived application to incorporate all of the layout options and controls that I would like to demonstrate. By necessity, this guide is more of a manual than a narrative.

After reading this guide, I would like you to be comfortable with creating a full application UI with `elm-ui`, and to understand all its capabilities and limitations.

The guide is divided into the following sections:

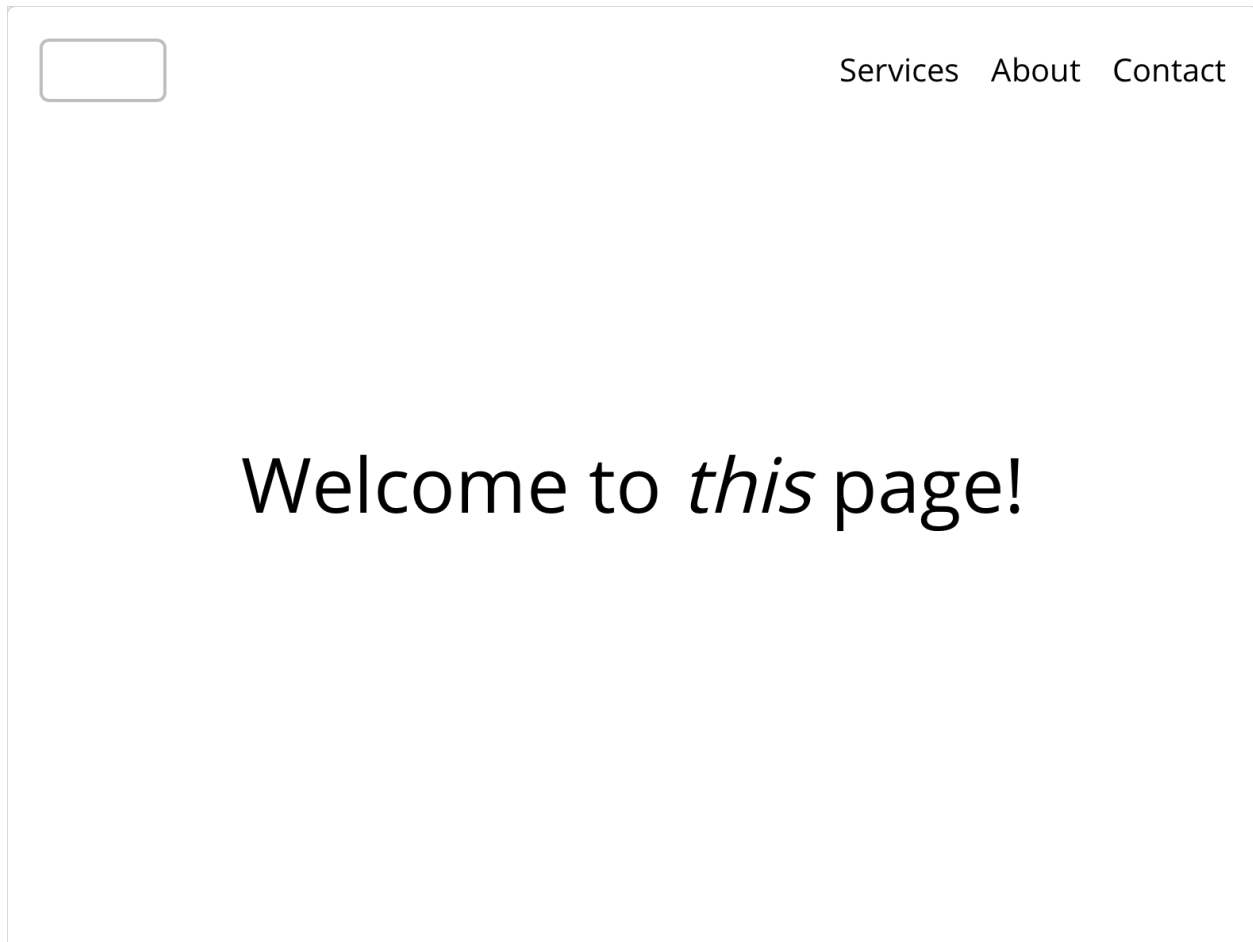
- Layouts
- Page elements and controls
- Typography
- More on styling
- Working with various types of content
- Organising UI code



## 2. Layouts

### 2.1 Getting started: title page layout

To see elm-ui in action, let's implement a simple title page layout:



There are a few things to note about this layout:

- the title text is centred both vertically and horizontally in the viewport

- the menu bar is fixed to the top of the viewport
- there is a left-aligned “logo” box on the left of the menu bar
- the menu items are right-aligned in the menu bar

First off, let's initialise the project:

```
$ elm init
$ elm install mdgriffith/elm-ui
```

Here is the code needed to implement this layout:

```
module TitleLayout exposing (..)

import Element exposing (..)
import Element.Border as Border
import Element.Font as Font
import Html exposing (Html)

menu : Element msg
menu =
    row
        [ width fill
        , padding 20
        , spacing 20
        ]
        [ el
            -- "logo" element
            [ width <| px 80
            , height <| px 40
            , Border.width 2
            , Border.rounded 6
            , Border.color <| rgb255 0xc0 0xc0 0xc0
            ]
            none
        , el [ alignRight ] <| text "Services"
        , el [ alignRight ] <| text "About"
        , el [ alignRight ] <| text "Contact"
        ]
```

```
main : Html msg
main =
  layout
    [ width fill, height fill, inFront menu ] <|
    el [ centerX, centerY, padding 50 ] <|
    paragraph
      [ Font.size 48, Font.center ]
      [ text "Welcome to "
      , el [ Font.italic ] <| text "this"
      , text " page!"
      ]
```

This code contains many of the key elm-ui elements, so let's go through it with a fine-tooth comb, starting with the imports:

```
import Element exposing (..)
import Element.Border as Border
import Element.Font as Font
import Html exposing (Html)
```

There are a number of modules in elm-ui. The main module is `Element` which contains a lot of `Element`-returning functions and layout attributes. Other attributes are split across several modules such as `Element.Border` and `Element.Font` in order to allow readable attribute specifications like `Font.italic` or `Border.width 3`. (Typically, you alias the module names like `import Element.Font as Font` so that your attribute lists read nicely.)

I also have to import `Html` for the main function. In more complex programs, you'll likely need `Browser` and `Html.Attributes` as well.

Next, let's break down `main`:

```

main : Html msg
main =
  layout -- convert top level element to `Html msg`
    [ width fill, height fill -- fill the width & height of viewport
    , inFront menu -- display the menu element fixed
                        -- to the viewport
    ] <|
    el -- an element with a single child
      [ centerX, centerY -- centre child element
        , padding 50 -- horizontally & vertically
                        -- add 50px of padding to contents
      ] <|
      paragraph -- display child elements inline
        [ Font.size 48 -- use 48px font for paragraph
          , Font.center -- centre align the children
        ]
        [ text "Welcome to " -- text node - note no attributes
          , el [ Font.italic ] -- wrap in an `el` to add attributes
              <| text "this"
          , text " page!"
        ]

```

In main, I used the `layout : List (Attribute msg) -> Element msg -> Html msg` function. Its main purpose is to convert from `Element msg` to `Html msg`. Typically you'll have just a single call of this function in your program, although it is possible to have multiple layouts if needed.

`Element msg` values are the main building block returned by `elm-ui` functions. You can think of an element as analogous to a `div` in HTML. Each element can have a number of attributes (`Attribute msg` values), aside from `text` element and empty element.

`el` returns an `Element` with a single child element.

Elements are composed with functions like `paragraph` which takes `List (Element msg)` and returns another `Element msg`.

The `paragraph` function arranges its child elements inline.

The call to `inFront` is a bit of an odd duck. It takes an `Element` as its argument but appears

inside the `layout` attribute list. I'm not sure whether this is for technical or aesthetic reasons, but it's easy enough to get used to when keeping in mind that an `inFront` element is displayed outside of the regular flow of child elements and so doesn't belong with the list of children.

### 2.1.1 The menu bar

The last part of this layout is the menu element:

```
menu : Element msg
menu =
  row                                     -- arrange children side by side
    [ width fill                         -- fill the width of container
      , padding 20                       -- pad contents at 20px
      , spacing 20                       -- keep 20px between child elements
    ]
    [ el                                 -- "logo" element
      [ width <| px 80
        , height <| px 40
        , Border.width 2                 -- 2px wide border
        , Border.rounded 6              -- round border with 6px radius
        , Border.color <| rgb255 0xc0 0xc0 0xc0 -- grey border
      ]
      none                               -- no content within element
      , el [ alignRight ] <| text "Services" -- right-align text item
      , el [ alignRight ] <| text "About"
      , el [ alignRight ] <| text "Contact"
    ]
```

`row` arranges its child elements side by side. As you will see later, there is also `column` and several other layout functions that group elements into different display configurations.

## 2.2 Going further: a chat page layout

Now let's look at a somewhat more complex layout – a chat page that resembles the Slack UI.

We'll begin with the smallest increment towards this layout:

```
module Main exposing (main)

import Element exposing (..)
import Element.Background as Background
import Element.Border as Border
import Element.Events exposing (..)
import Element.Font as Font
import Element.Input as Input
import Html exposing (Html)

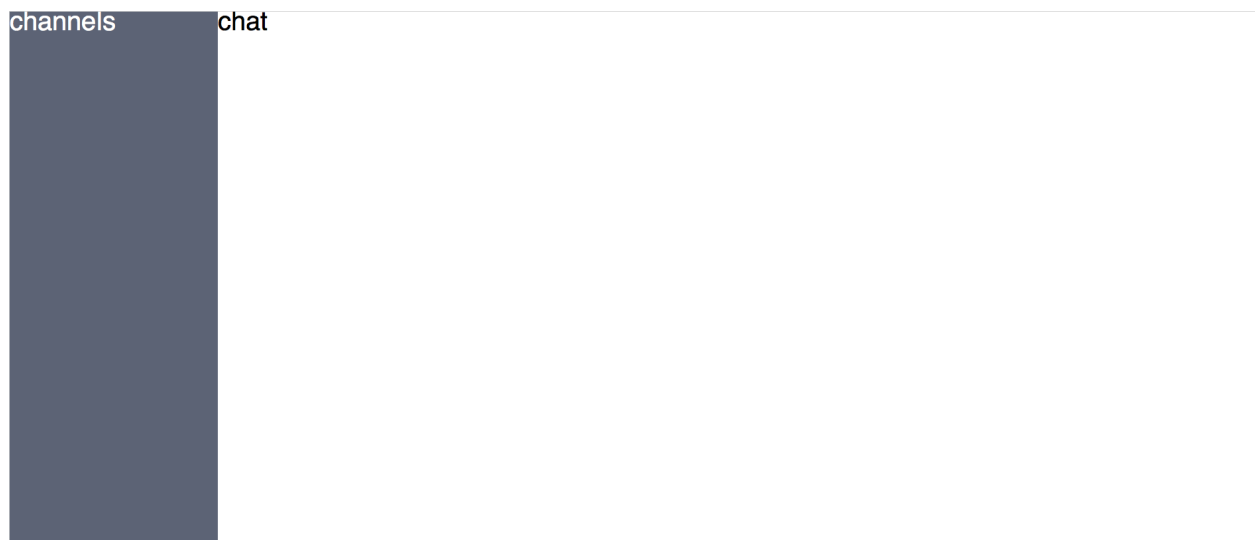
channelPanel : Element msg
channelPanel =
    column
        [ height fill
        , width <| fillPortion 1
        , Background.color <| rgb255 92 99 118
        , Font.color <| rgb255 255 255 255
        ]
        [ text "channels" ]

chatPanel : Element msg
chatPanel =
    column [ height fill, width <| fillPortion 5 ]
        [ text "chat" ]

main : Html msg
main =
    layout [ height fill ] <|
        row [ height fill, width fill ]
            [ channelPanel
            , chatPanel
            ]
```

Here is the page produced by this code:





Let's examine main first:

```
main : Html msg
main =
  layout [] <|
    row [ height fill, width fill ]
      [ channelPanel
        , chatPanel
      ]
```

As row arranges its child elements side by side, it will result in the channel panel on the left and chat messages on the right.

### 2.2.1 Channel and chat panels

The definitions of channelPanel and chatPanel use column which arranges its child elements vertically:

```
channelPanel : Element msg
channelPanel =
  column
    [ height fill
      , width <| fillPortion 1
      , Background.color <| rgb255 92 99 118
      , Font.color <| rgb255 255 255 255
      ]
    [ text "channels" ]

chatPanel : Element msg
chatPanel =
  column [ height fill, width <| fillPortion 5 ]
    [ text "chat" ]
```

Both of them need to fill the full height of the parent element. As far as width goes, I allocated 1 part of the total width to the channel list and 5 parts to the chat by using `fillPortion`. The total number of parts that the parent width is divided into is the sum of all `fillPortion` part counts across the sibling elements. This approach may take a bit of getting used to after using percentages in CSS, but is quite intuitive.

You'll also note that I specified the background and font colors for the channel panel. The attribute definitions are split into different modules for readability, so we need to import two different modules for the colour attributes:

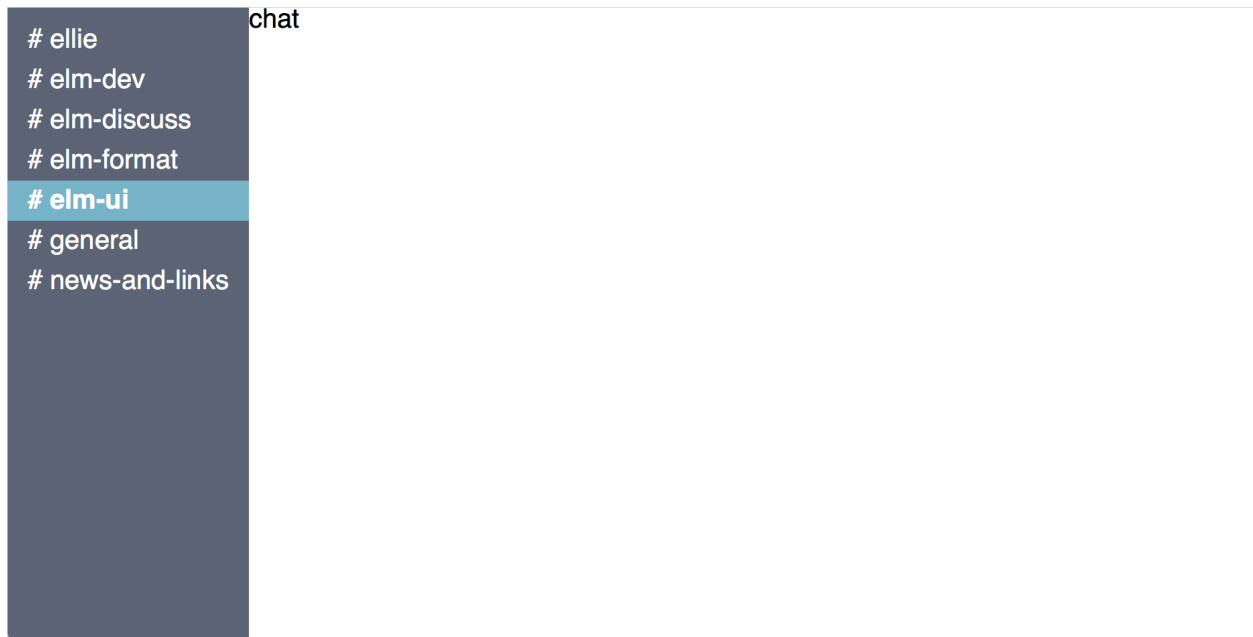
```
import Element.Background as Background
import Element.Border as Border
```

`row` and `column` functions are the two workforces of layout in `elm-ui`. Combining them is surprisingly versatile and allows you to reproduce, for example, most of the functionality of CSS Grid, including nested grids. To me, defining grids this way is more intuitive. There might be a bit more nesting, but it's easy to turn the idea of a layout into an implementation, and easy to visualise what exactly is going on.

There are several more layout functions available, as you will see later.

### 2.2.2 Filling in the channel list

Next, let's show a list of channels, highlighting the active channel:



Here is how I can implement this:

```
channelPanel : List String -> String -> Element msg
channelPanel channels activeChannel =
    let
        activeChannelAttrs =
            [ Background.color <| rgb255 117 179 201, Font.bold ]

        channelAttrs =
            [ paddingXY 15 5, width fill ]

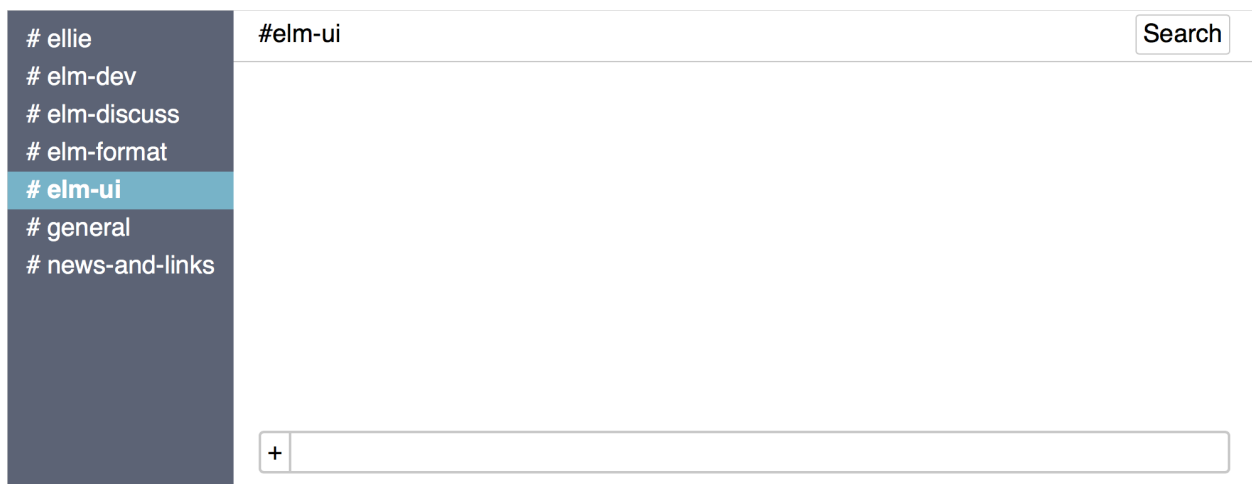
        channelEl channel =
            el
                (if channel == activeChannel then
                    activeChannelAttrs ++ channelAttrs
                else
                    channelAttrs
                )
            <|
                text ("# " ++ channel)
```

```
in
column
  [ height fill
    , width <| fillPortion 1
    , paddingXY 0 10
    , Background.color <| rgb255 92 99 118
    , Font.color <| rgb255 255 255 255
    ]
<|
List.map channelEl channels
```

The main component of this function is `channelEl` in the `let`. I specify different attributes based on whether the channel is the active channel. This function is mapped over the list of channels.

### 2.2.3 Header and footer sections in the channel

Next, I can start filling in parts of the content area:



The corresponding code looks like this:

```

chatPanel : String -> Element msg
chatPanel channel =
  let
    header =
      row
        [ width fill
          , paddingXY 20 5
          , Border.widthEach { bottom = 1, top = 0, left = 0, right = 0 }
          , Border.color <| rgb255 200 200 200
        ]
        [ el [] <| text ("#" ++ channel)
          , Input.button
              [ padding 5
                , alignRight
                , Border.width 1
                , Border.rounded 3
                , Border.color <| rgb255 200 200 200
              ]
              { onPress = Nothing
                , label = text "Search"
              }
        ]

    messagePanel =
      column [] []

    footer =
      el [ alignBottom, padding 20, width fill ] <|
        row
          [ spacingXY 2 0
            , width fill
            , Border.width 2
            , Border.rounded 4
            , Border.color <| rgb255 200 200 200
          ]
          [ el
              [ padding 5
                , Border.widthEach { right = 2, left = 0, top = 0, bottom = 0 }
                , Border.color <| rgb255 200 200 200
                , mouseOver [ Background.color <| rgb255 86 182 139 ]
              ]
            <|
              text "+"
          ]

```

```
        , el [ Background.color <| rgb255 255 255 255 ] none
      ]
in
column [ height fill, width <| fillPortion 5 ]
  [ header
    , messagePanel
    , footer
  ]
```

At first glance, this might look like a lot of code, however note that it's really straightforward, and most of it is simply layout and styling attributes. At the bottom of the function, it's very clearly stated that we have vertically arranged header, messages and footer.

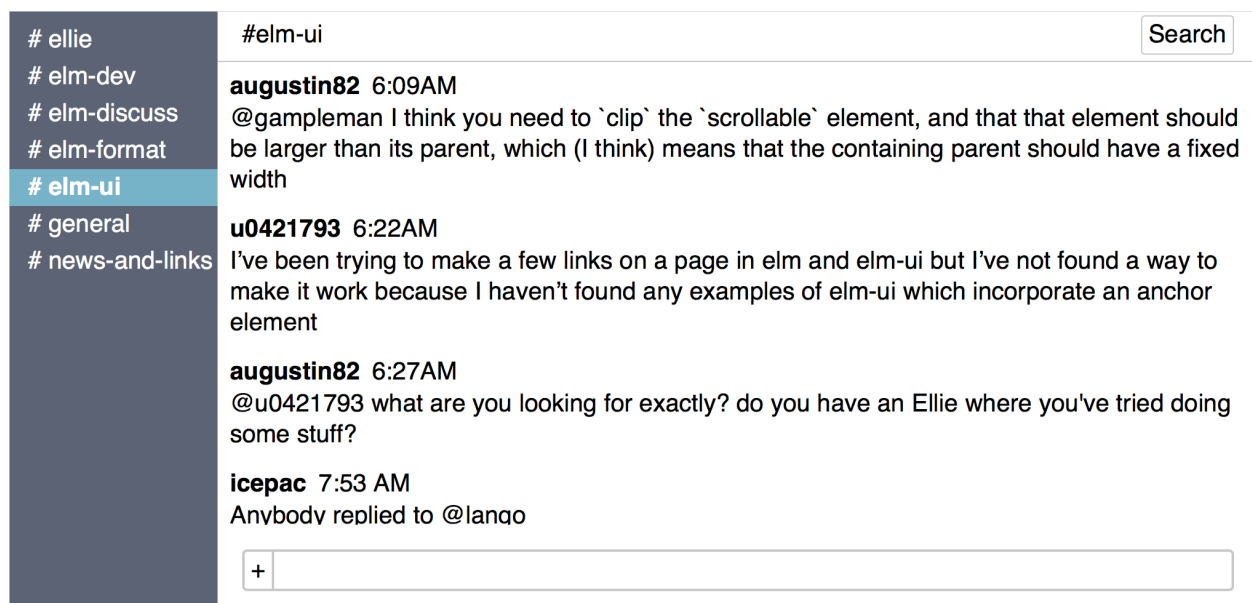
The definitions of `header` and `footer` are very readable, and conceptually simple. I'm still combining rows and columns, with a sprinkling of padding and spacing, plus some visual styling like colours.

All I need to do to attach the footer to the bottom is to give it the `alignBottom` attribute.

## 2.2.4 Messages

The last thing we need to implement is the message panel. We want it to take up the available screen space, and to scroll when there are more messages than fit on the screen:





We have to make `chatPanel` take a list of messages as its argument, and to flesh out `messagePanel` in its `let` clause:

```
type alias Message =
  { author : String, time : String, text : String }

...

chatPanel : String -> List Message -> Element msg
chatPanel channel messages =
  let
    header =
      ...

    messageEntry message =
      column [ width fill, spacingXY 0 5 ]
        [ row [ spacingXY 10 0 ]
          [ el [ Font.bold ] <| text message.author, text message.time ]
          , paragraph [] [ text message.text ]
        ]

    messagePanel =
      column [ padding 10, spacingXY 0 20, scrollbarY ] <|
        List.map messageEntry messages
```

```

        footer =
            ...
    in
    column [ height fill, width <| fillPortion 5 ]
        [ header
          , messagePanel
          , footer
        ]

```

In defining `messageEntry`, I used `paragraph` which you previously encountered in the title page layout. This function lays out its children as wrapped inline elements. In this case there's only one child - the message text, but I still need it to wrap.

Also note that the column in `messagePanel` has the `scrollbarY` attribute – this is what prevents it from pushing the footer beyond the bottom edge of the viewport.

There is one more change required to constrain the whole layout to the viewport (rather than having the messages extend it further down past the fold). The top level layout needs to have `height fill` in its attributes:

```

main : Html msg
main =
    layout [ height fill ] <|
        row [ height fill, width fill ]
            [ channelPanel sampleChannels sampleActiveChannel
              , chatPanel sampleActiveChannel sampleMessages
            ]

```

And that's it! The whole layout took 150 lines of code – not bad for a non-trivial layout, and it's very clear code to boot, with no HTML or CSS in sight.

## 2.3 Layout functions

Now that we've seen the basics of making a layout with `elm-ui`, let's take a more systematic look at the available layout functions.

### 2.3.1 Basic layout

You have already seen basic layout functions:

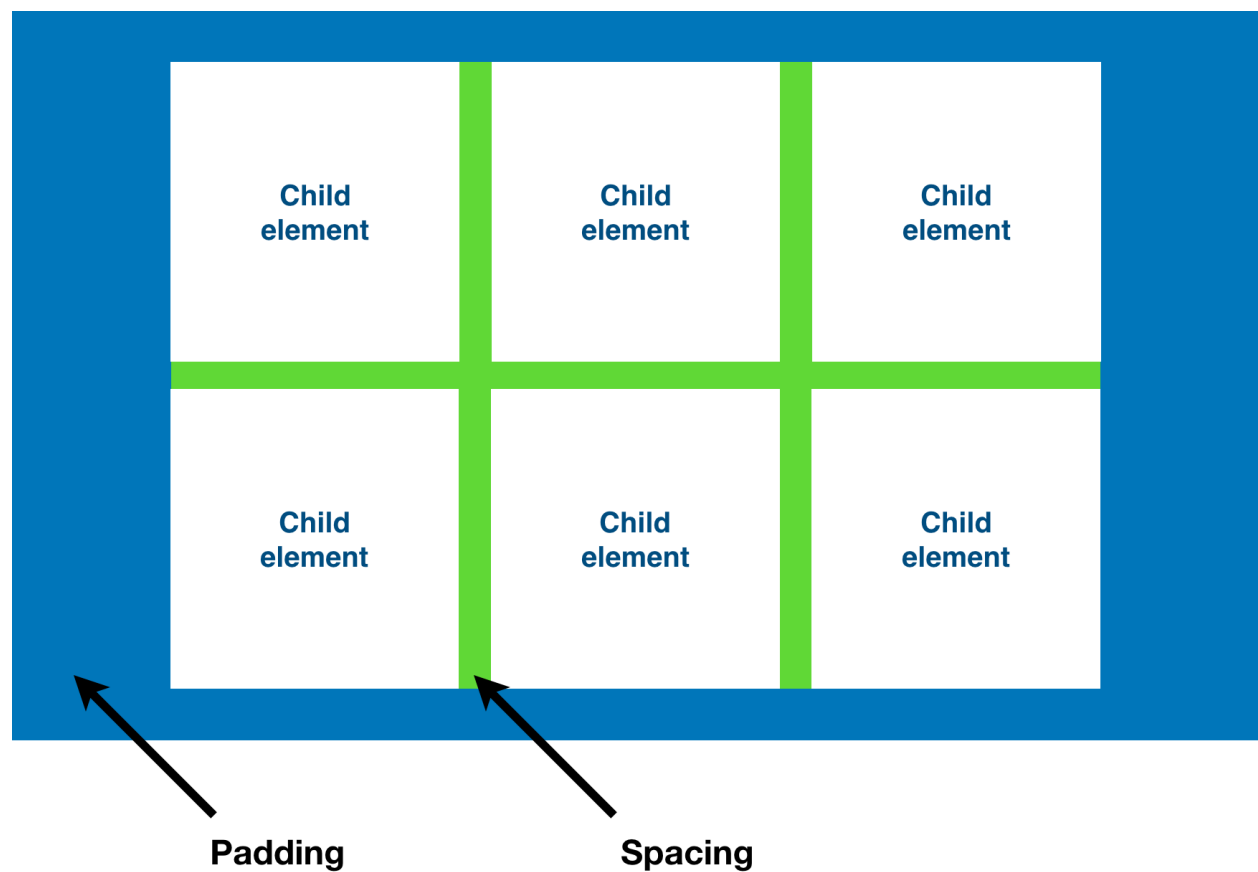
- `el` – the basic building block of layouts with one child element (often `text`)
- `text` – a text element without wrapping
- `none` – an empty element

To group elements together, you can use:

- `row` – arranges its child elements side by side
- `column` – stacks its child elements vertically
- `wrappedRow` – arranges its child elements side by side but wraps them if they use up too much space (similar to `inline-block` in CSS).

### 2.3.2 Padding and spacing

It's worth noting that the `padding` attribute works differently from CSS. `elm-ui` does away with the rather troublesome concept of margins and padding found in CSS. Instead, you can specify internal padding for an element, and you can also specify the spacing between its child elements:



The padding attribute can be specified in three ways:

- padding pads the content of an element with a given number of pixels on each side.
- paddingXY allows you to specify horizontal and vertical padding separately
- paddingEach is for when you need different paddings on each side.

For spacing between elements, in most cases you can just use spacing but with layouts like `wrappedRow` and `textColumn`, you may want to set different horizontal and vertical spacing with `spacingXY`.

There is also the `spaceEvenly` attribute, which will distribute the children within the parent element with equal spacing between them.

### 2.3.3 Sizing

There are several way of specifying element dimensions:

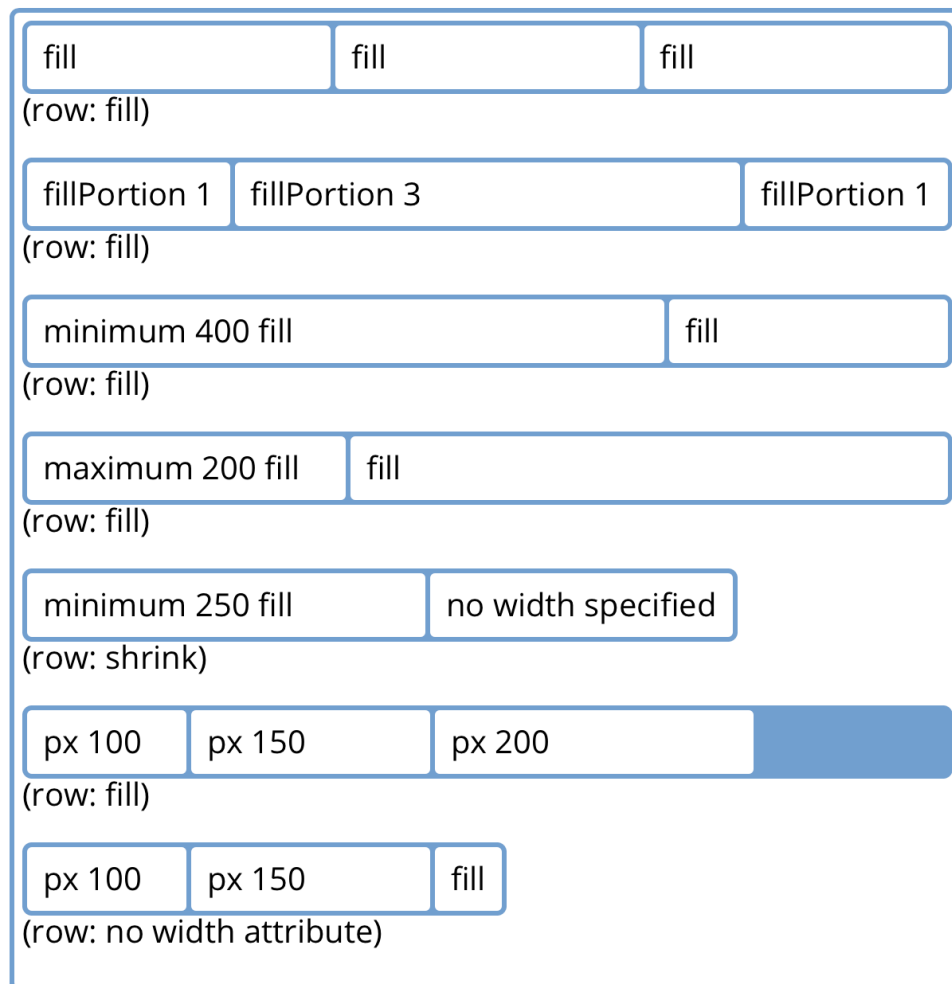
- `px` for a fixed number of pixels
- `fill` to fill the available space or share it evenly with other elements set to `fill` space
- `fillPortion` to fill the specified portion of element
- `shrink` to make the size of the element match the size of its contents.

Child elements can overflow the parent. For example, if the total width of the children is larger than the width of the parent row, they will overflow. Child elements don't expand the dimensions of the parent.

Additionally, you can constrain width and height by using maximum and minimum functions, for example:

- `width <| minimum 400 fill`
- `width <| maximum 250 <| fillPortion 1`

Here is an illustration of some combinations of width attributes on `els` inside a row:



### 2.3.4 Transparency

Sometimes, you may need to hide an element without affecting the surrounding layout. In that case, you can use the transparent attribute:



```
Input.button
[ transparent True
, padding 20
, Border.width 2
, Border.color <| rgb255 0x50 0x50 0x50
]
{ onPress = Nothing
, label = text "Button"
}
```

A transparent element is supposed to stop receiving input, however as of v1.1.7, due to a bug the element continues to receive input, so you need to disable input handling yourself, eg by setting `onPress = Nothing`.

It's also possible to set the opacity of an element using the `alpha` attribute:

```
column [ width fill ]
[ el [ width fill, height <| px 30, Background.color bluish ] <|
  text "First element"
, el [ alpha 0.5, width fill, height <| px 30, Background.color bluish ] <|
  text "Second element - alpha 0.5"
, el [ width fill, height <| px 30, Background.color bluish ] <|
  text "Third element"
]
```

First element

Second element - alpha 0.5

Third element

### 2.3.5 Alignment

For horizontal alignment, we have:

- `alignLeft`

- `centerX`
- `alignRight`

For vertical alignment, we have:

- `alignTop`
- `centerY`
- `alignBottom`

An element can have both horizontal and vertical alignment. Note that for text alignment, there are different attributes available in the `Font` module. You can read about them in the section on typography.

There are a couple of things worth noting about the behaviour of these attributes. An element with an alignment towards one of the sides of the parent (eg `alignRight`) will push other elements which are closer to that boundary. This is illustrated below.

Centring an element horizontally or vertically means centring it *in the available space* rather than in the parent container. This is also shown in the image below. Compare second example (`centerX` in available space) to the last example (`centerX` in container):



Centring relative to a parent element requires introducing an extra level of nesting. Compare these examples:

```
-- Centre in available space
row [ width fill ]
  [ el [] <| text "no align"
    , el [ centerX ] <| text "centerX"
    , el [] <| text "no align"
    , el [] <| text "no align"
  ]

-- Centre relative to the parent element
row [ width fill ]
  [ el [ width <| fillPortion 2 ] <| box [] <| text "no align"
    , el [ width <| fillPortion 1 ] <| box [ centerX ] <| text "centerX"
    , row [ width <| fillPortion 2 ]
      [ box [ alignRight ] <| text "alignRight"
        , box [] <| text "no align"
      ]
  ]
```

Centring relative to the parent requires an extra level of nesting. First you have to break up the elements into groups which are sized symmetrically around the central element. Then, you can align the elements inside those groups as appropriate, with `centerX` applied to the element inside the central container.

### 2.3.6 Dealing with content overflow

When the content overflows, there are three options:

- just let it happen
- clip it at the container boundaries
- allow the container to scroll.

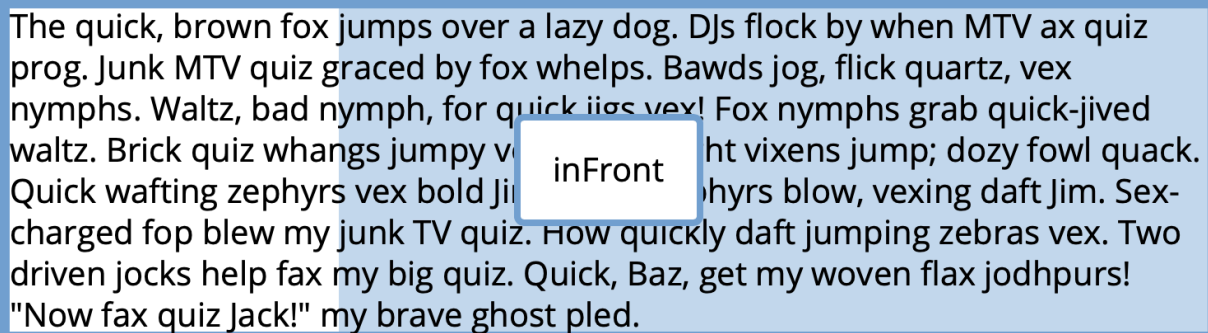
The first outcome will happen by default with `elm-ui`. To clip content, you can add the `clip` attribute (or, if you want to only clip overflow in one direction, `clipX` or `clipY`). Note that you may need to set the dimensions on an element in order for the clipping to apply. Without dimensions, it's unclear at what point the clipping should occur.

To allow scrolling, you can enable one or two scrollbars by adding the `scrollbars`, `scrollbarX` or `scrollbarY` attributes.

### 2.3.7 Placing elements near, above, or below other elements

Adjacent placement is achieved by way of converting elements into attributes. You've already seen one example of that at the start of the chapter, in the implementation of the menu for the title page layout with `inFront`.

Elements can also be placed behind content, but above the background of an element:



The quick, brown fox jumps over a lazy dog. DJs flock by when MTV ax quiz prog. Junk MTV quiz graced by fox whelps. Bawds jog, flick quartz, vex nymphs. Waltz, bad nymph, for quick jigs vex! Fox nymphs grab quick-jived waltz. Brick quiz whangs jumpy v for vixens jump; dozy fowl quack. Quick wafting zephyrs vex bold Jim. Quick zephyrs blow, vexing daft Jim. Sex-charged fop blew my junk TV quiz. How quickly daft jumping zebras vex. Two driven jocks help fax my big quiz. Quick, Baz, get my woven flax jodhpurs! "Now fax quiz Jack!" my brave ghost pled.

The white stripe behind the text on the left is created with `behindContent`, and the element sitting on top of the content is added with `inFront`:

```
el
  [ inFront <|
    el
      [ centerX
        , centerY
        , padding 20
        , Border.width 4
        , Border.color blue
        , Border.rounded 6
        , Background.color <| rgb255 255 255 255
      ]
    <|
      text "inFront"
    , behindContent <|
```

```

    el
      [ alignTop
      , height fill
      , width <| px 200
      , Background.color <| rgb255 255 255 255
      ]
      none
    , centerX
    , centerY
    , Border.width 10
    , Border.color blue
    , Border.rounded 6
    , Background.color <| rgba255 0x72 0x9F 0xCF 0.4
  ]
<|
  paragraph [] [ text sampleText ]

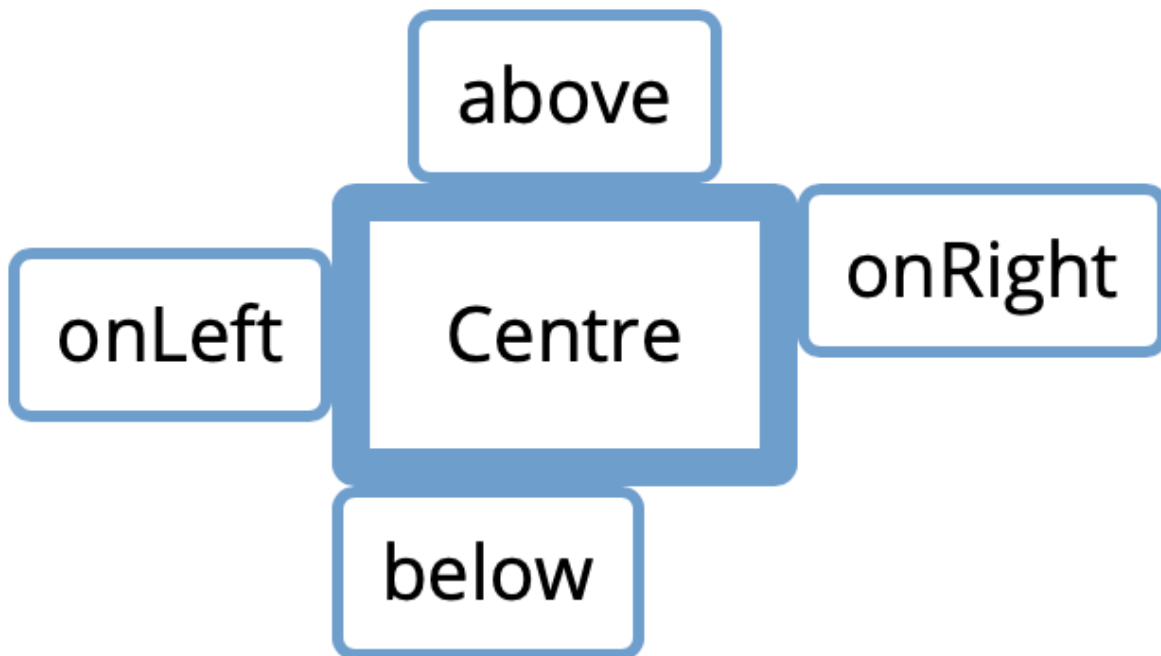
```

These elements can be positioned and sized relative to the main element, as the above example shows.

There's a special case if you add an `inFront` element to a layout. `elm-ui` will place the element in a fixed position relative to the viewport, which is useful for overlaid menus and dialogs.

Elements can additionally be positioned on any of the four sides of an element:





```
blue = rgb255 0x72 0x9F 0xCF
```

```
box : List (Attribute msg) -> Element msg -> Element msg
```

```
box attrs =
```

```
  el
```

```
    ([ Background.color <| rgb255 255 255 255
```

```
      , padding 10
```

```
      , Border.rounded 6
```

```
      , Border.width 3
```

```
      , Border.color blue
```

```
    ]
```

```
      ++ attrs
```

```
  )
```

```
main : Html msg
```

```
main =
```

```
  layout [ width fill, height fill ] <|
```

```
    el
```

```
      [ centerX
```

```
      , centerY
```

```
      , above <| box [ centerX ] <| text "above"
```

```
      , below <| box [] <| text "below"
```

```

    , onRight <| box [ alignTop ] <| text "onRight"
    , onLeft <| box [ centerY ] <| text "onLeft"
  ] <|
box
  [ Border.width 10
  , padding 20
  ] <| text "Main"

```

elm-ui currently ignores the borders of the main element when positioning adjacent elements, which is why I had to add a level of nesting to the “Main” element, and attach nearby elements to the outside el.


As with elements placed above or behind content, the adjacent elements can have their own attributes (eg `centerX`) applied to position them relative to the main element.

Attributes with adjacent elements can be nested: for example, an element given to `onLeft` can have its own `onLeft` attribute, which can be useful for deeply nested menus or other hierarchically organised elements.

## 2.4 Translation, rotation and scaling

The placement of any element can be “adjusted” by applying a horizontal or vertical offset, rotating or scaling it.

An element can be offset using one of the four attribute functions `moveUp`, `moveDown`, `moveLeft`, `moveRight`. Here, for example, the second of three identical elements in a column is shifted out of its position:



```
moveLeft 10, moveUp 20
```

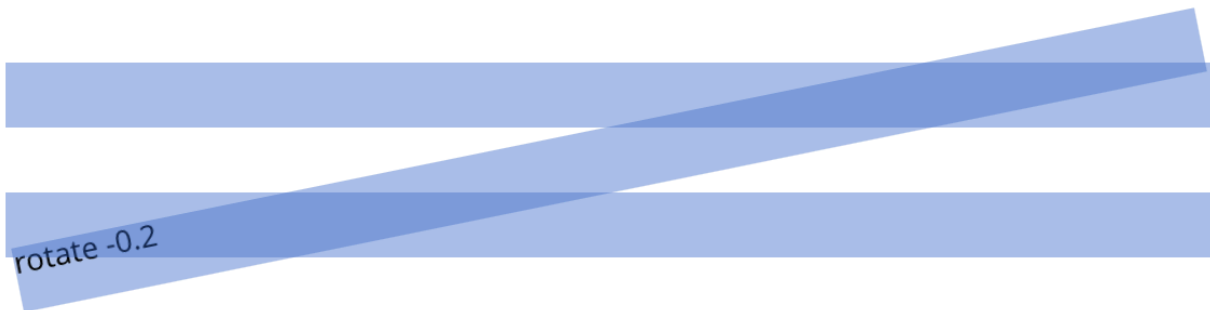
```

column [ width fill ]
  [ el [ width fill, height <| px 30, Background.color bluish ] none
    , el
      [ moveLeft 10
        , moveUp 20
        , width fill
        , height <| px 30
        , Background.color bluish
      ]
    <|
      text "moveLeft 10, moveUp 20"
    , el [ width fill, height <| px 30, Background.color bluish ] <| none
  ]

```

Each of these functions takes an offset in pixels (which can be positive or negative).

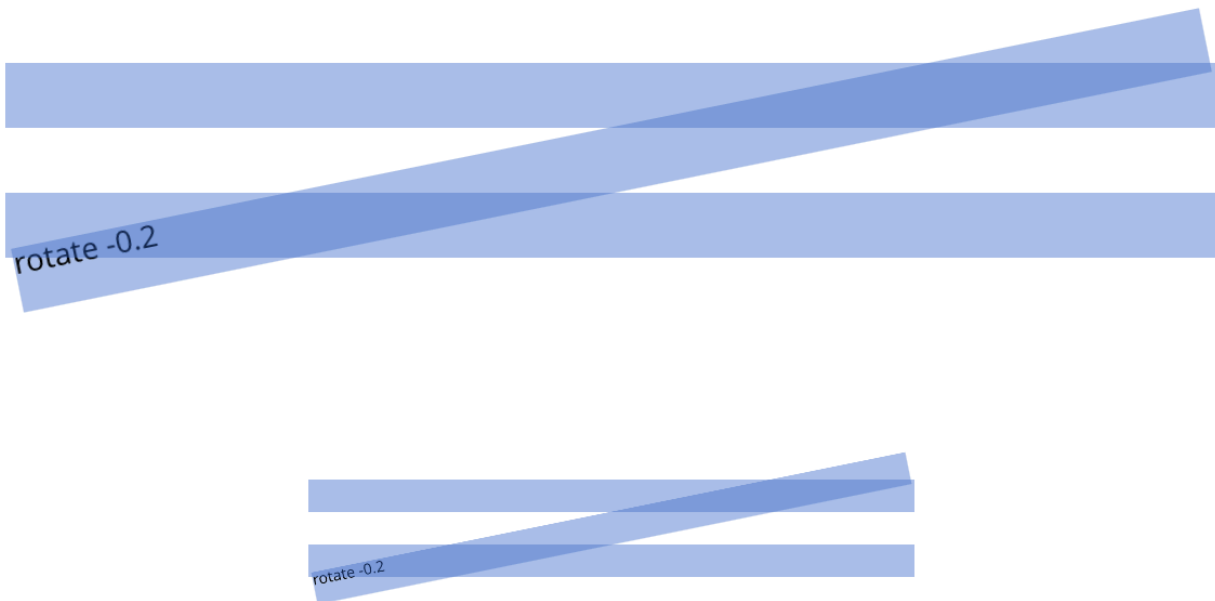
An element can be rotated by adding a rotate attribute which specifies the angle in radians (2 \* pi radians make 360 degrees):



```
column [ width fill ]
  [ el [ width fill, height <| px 30, Background.color bluish ] none
    , el
      [ rotate -0.2
        , width fill
        , height <| px 30
        , Background.color bluish
        ]
    <|
      text "rotate -0.2"
    , el [ width fill, height <| px 30, Background.color bluish ] none
  ]
```

The adjustment happens after the non-adjusted element is placed, so it doesn't affect the rest of the layout.

Finally, an element can be scaled together with its contents:



Here, the bottom column with three elements is defined the same as the top column, but additionally has a `scale 0.5` attribute.

When scaling up (using values greater than 1), the element may become blurry, so you may be better off using the larger version of the element as the “original” and scaling that down,

rather than going the other way and scaling up.

## 2.5 Responsive layouts

elm-ui gives you a powerful tool for making your layout responsive: Elm itself. As views are fully defined in Elm, you can change them in any way you like in response to viewport size changes. For example, you can switch from showing elements in a row to showing them in a column.

elm-ui provides one helper function to help determine what kind of viewport you're dealing with: `classifyDevice`.

This function takes a record with `width` and `height` fields specifying the size of the viewport in pixels, and returns a `Device` record which tells you the device class (phone, tablet, desktop or "big desktop") and screen orientation (portrait or landscape).

Alternatively, you can simply subscribe to window resize events, and write conditional view code directly based on dimensions:

```
import Browser.Events exposing (onResize)

subscriptions : Model -> Sub Msg
subscriptions _ =
    onResize UserResizedWindow

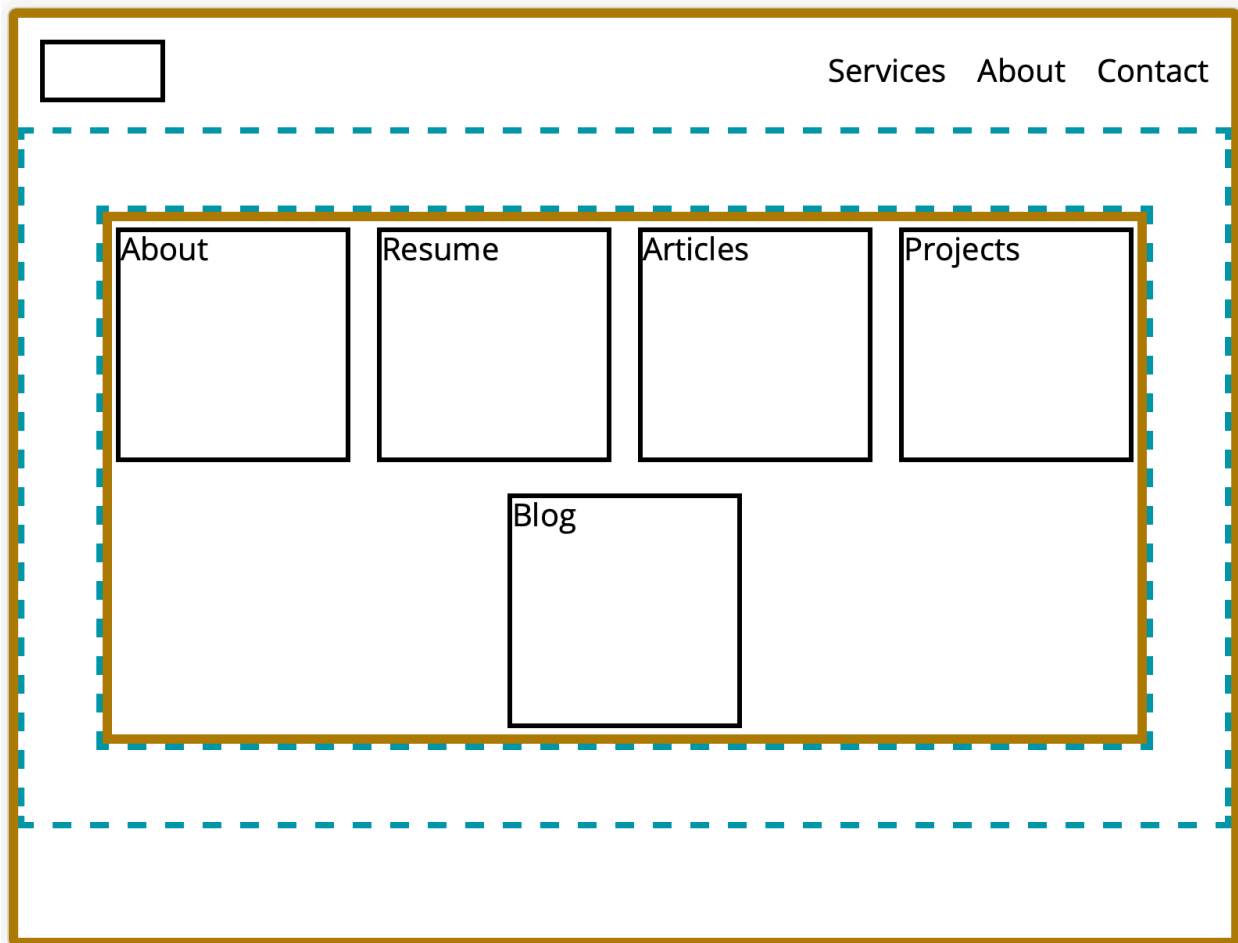
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        ...
        UserResizedWindow width height ->
            ( { model | windowSize = { height = height, width = width } }, Cmd.none )
        ...
```

A nuance with responsive layouts is that in order to render the appropriate layout, you need to know the viewport size when the app first loads. To achieve that, you need to pass in dimensions via flags:

```
<script>
let app = Elm.Main.init({
  flags: {
    windowHeight: window.innerHeight,
    windowWidth: window.innerWidth
  }
})
</script>
```

## 2.6 Debugging

Sometimes you'll have an issue with your layout where it's not clear how elements are sized. In those situations, you can use the `explain` attribute to highlight element borders:



```
main : Html msg
main =
  layout [ explain Debug.todo, width fill, height fill, inFront menu ] <|
    el [ centerX, centerY, padding 50 ] <|
      wrappedRow [ explain Debug.todo, spacing 20 ]
        [ box "About"
          , box "Resume"
          , box "Articles"
          , box "Projects"
          , box "Blog"
        ]
```

This attribute makes borders visible on the element it's applied to as well as its immediate children. You have to pass it `Debug.todo`, which is a trick to make it impossible to leave `explain` in release builds.

## 2.7 Escape hatches

It's always useful to be able to have escape hatches in case something you need to do isn't possible with the existing capabilities of the package. `elm-ui` provides two escape hatch mechanisms:

- `Element.html` takes a `Html msg` value and turns it into `Element msg`, allowing you to put it into an `elm-ui` layout
- `Element.htmlAttribute` takes an `Html.Attribute msg` and turns it into an `elm-ui` attribute.

For example, you might want to add an ID to an element. You can do it like this:

```
el [ Element.htmlAttribute <| Html.Attributes.id "my-element" ] <| text "Element with ID"
```

Another example is if you want to display some Markdown with the help of `elm-explorations/markdown`. That package happens to turn it into `Html msg` values, so in order to include the rendered Markdown in a layout, you need `Element.html`:

```
paragraph []  
  [ Element.html <|  
    Markdown.toHtmlWith mdOptions [ Attr.class "markdown" ] markdownStr  
  ]
```

One caveat is that none of the `elm-ui` styling will apply here, and you will need to either include inline styles or add a stylesheet for the “raw” HTML bits.



## 3. Page elements and controls

A web page can include elements such as links or images. `elm-ui` has these functions:

- `link` and a few specialised versions of links
- `image` for displaying images
- `table` for showing tabular data

`elm-ui` also includes versions of the standard browser input controls:

- `button`
- `checkbox`
- `radio`
- `text` for a single-line text input, as well as a bunch of inputs that work with browser autofill, such as `username` and `email`
- `multiline` for a multiline text input
- `slider`

Note that `dropdown` isn't in this list. That's likely because dropdowns are often a UI anti-pattern, as they are difficult to use and can usually be replaced with a list of selectable options. However, if you need one, you can still implement it using the primitives provided by `elm-ui`, or you could use a package such as [PaackEng/elm-ui-dropdown](https://package.elm-lang.org/packages/PaackEng/elm-ui-dropdown/latest/)<sup>1</sup>.

By default, the controls are unstyled. For example, here is the default rendering of a button and a checkbox:

---

<sup>1</sup><https://package.elm-lang.org/packages/PaackEng/elm-ui-dropdown/latest/>

Unstyled button

☒ Unstyled checkbox

It's up to you to add the necessary styles.

Unlike HTML, elm-ui doesn't include any grouping mechanisms like `<form>` or `<fieldset>`.

Another aspect of user input is events from input devices. elm-ui includes attributes for adding messages generated from mouse and focus events. These can be added to any elements.

Let's take a closer look at these page elements, controls, and events one by one.

## 3.1 Links

What web application is complete without links? An elm-ui link needs a URL and a label, which is an Element:

```
link [] { url = "https://example.com", label = text "Unstyled link" }
```

You can add attributes to style it:

```
link [ Font.bold, Font.underline, Font.color blue ]  
  { url = "https://example.com", label = text "Styled link" }
```

The label can be any element, for example an image:

```
link []
  { url = "https://example.com"
  , label =
      image []
        { src = "https://picsum.photos/200/100"
        , description = "Image link"
        }
  }
```

There are three special kinds of links. `newTabLink` opens in a new browser tab:

```
newTabLink [ Font.bold, Font.underline, Font.color blue ]
  { url = "https://example.com", label = text "New tab link" }
```

`download` and `downloadAs` trigger file downloads:

```
download [ Font.bold, Font.underline, Font.color blue ]
  { url = "/elm.json", label = text "Download file" }

downloadAs [ Font.bold, Font.underline, Font.color blue ]
  { url = "/elm.json", filename = "renamed.json", label = text "Download renamed file" }
```

## 3.2 Images

An image just needs a source and a description (for assistive technologies):

```
image []
  { src = "https://picsum.photos/300/200"
  , description = "An image"
  }
```

A 2x resolution image can be obtained by halving display width and height:

```
image [ width <| px 300, height <| px 200 ]  
  { src = "https://picsum.photos/600/400"  
    , description = "A 2x image"  
  }
```

Sometimes images need to be clipped, like in the popular round profile photos. This can be achieved by wrapping an image in an element with clipped contents and a circular border:

```
el [ Border.rounded 150, clip ] <|  
  image []  
    { src = "https://picsum.photos/300/300"  
      , description = "Circular image"  
    }
```



It's also possible to use images as background. The Background module includes several attribute options:

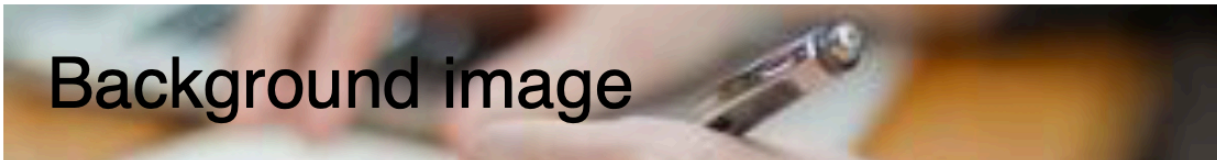
- `Background.image` for an image that's resized proportionally to the element, with overflow cropped
- `Background.uncropped` for an image that's resized to fit into the element, without cropping

- `Background.tiled` for an image that's tiled both vertically and horizontally
- `Background.tiledX` for an image that's tiled horizontally
- `Background.tiledY` for an image that's tiled vertically

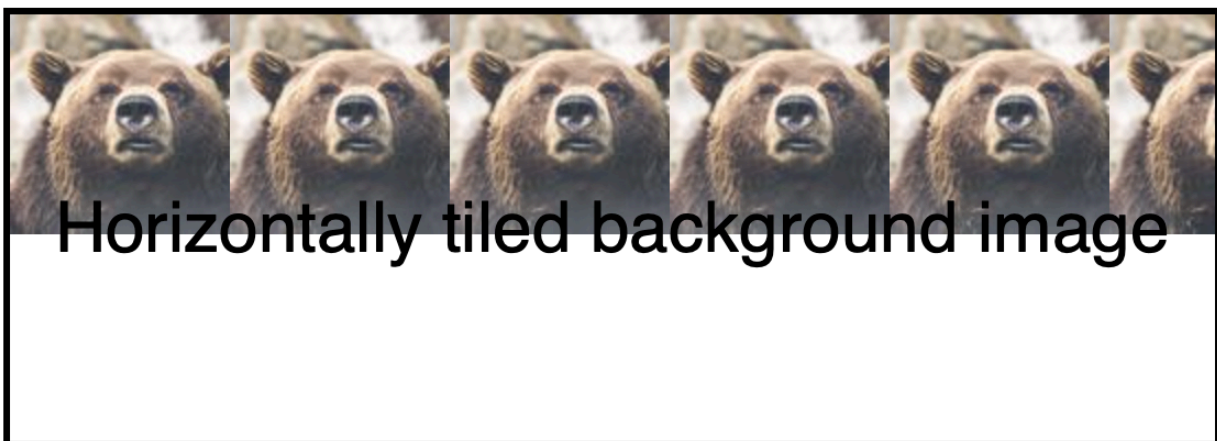
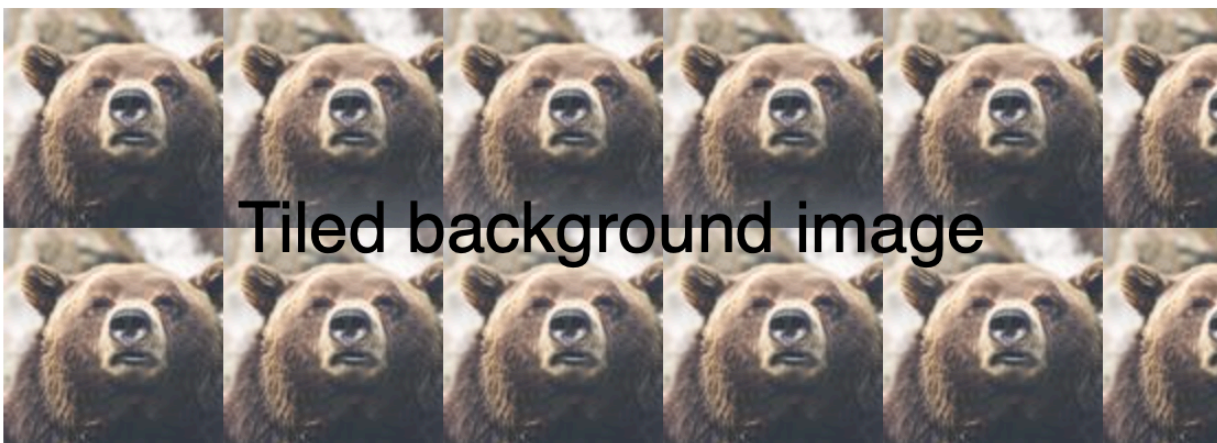
These attributes produce backgrounds shown below:

```
column []
  [ el
    [ Background.image "https://picsum.photos/300/300"
      , width fill
      , padding 20
      , Font.size 32
    ]
    <|
    text "Background image"
  , el
    [ Background.uncropped "https://picsum.photos/300/300"
      , width fill
      , padding 20
      , Font.size 32
    ]
    <|
    text "Uncropped background image"
  , el
    [ Background.tiled "https://picsum.photos/100/100"
      , width fill
      , height <| px 200
      , padding 20
      , Font.size 32
    ]
    <|
    el [ centerX, centerY ] <|
      text "Tiled background image"
  , el
    [ Background.tiledX "https://picsum.photos/100/100"
      , width fill
      , height <| px 200
      , padding 20
      , Border.width 3
      , Font.size 32
    ]
  ]
```

```
<|  
  el [ centerX, centerY ] <|  
    text "Horizontally tiled background image"  
]
```



Uncropped background image



### 3.3 Tables

`Element.table` helps you display a list of records in tabular format:

```

colors : List { color : Color, name : String }
colors =
  [ { color = black, name = "Black" }
  , { color = blue, name = "Blue" }
  , { color = green, name = "Green" }
  , { color = orange, name = "Orange" }
  , { color = red, name = "Red" }
  ]

colorTable : Element msg
colorTable =
  let
    headerAttrs =
      [ Font.bold
      , Font.color blue
      , Border.widthEach { bottom = 1, top = 0, left = 0, right = 0 }
      , Border.color black
      ]
  in
    table [ width shrink, spacing 10 ]
      { data = colors
      , columns =
        [ { header = el headerAttrs <| text "Color name"
          , width = fillPortion 2
          , view = .name >> text >> el [ centerY ]
          }
        , { header = el headerAttrs <| text "Color sample"
          , width = fillPortion 1
          , view =
            \rec ->
              el
                [ width fill
                , height <| px 40
                , Background.color rec.color
                ]
              none
          }
        ]
      }

```

```
    ]  
  }  
  
main : Html msg  
main =  
  layout [ width fill, height fill, padding 50 ] colorTable
```

## Color name

---

Black

Blue

Green

Orange

Red

## Color sample



To create a table, you need to provide `table` with a list of attributes (one useful attribute is `spacing` as it controls spacing between cells), as well as a record with two fields: `data` and `columns`. `data` is a list of records which can have any fields. `columns` is a list of records which define how each column is going to be rendered.

Each column record needs to have three fields:

- `header` which is an `Element` that will be the column header
- `width` which determines the width of the column



- view which is a function that takes a record from your data list and turns it into Element

Sometimes, you might need to know the row number when rendering cells. In that situation, you can use `indexedTable`. The only difference from `table` is that the `view` function in column definitions takes the row index in addition to the record: `view : Basics.Int -> record -> Element msg`.

## 3.4 Buttons

A button is created using `Input.button`, which requires supplying an element for the button label, and an optional message handler. `elm-ui` buttons are implemented with `<div>` rather than `<button>`. By default, a button is unstyled so it looks like plain text:

```
Input.button [] { onPress = Nothing, label = text "Unstyled button" }
```

### Unstyled button

With a few styles, it can be made to look more like a traditional button. You can include a hover style and a focus style to provide a better interaction:

```
Input.button
  [ padding 20
  , Border.width 2
  , Border.rounded 16
  , Border.color <| rgb255 0x50 0x50 0x50
  , Border.shadow { offset = ( 4, 4 ), size = 3, blur = 10, color = grey }
  , Background.color blue
  , Font.color white
  , mouseOver
    [ Background.color white, Font.color black ]
  , focused
    [ Border.shadow { offset = ( 4, 4 ), size = 3, blur = 10, color = blue } ]
  ]
{ onPress = Just UserPressedButton
, label = text "Button with focus style"
}
```

## Button with focus style

If you set the `onPress` handler to `Nothing`, it doesn't disable the button or change its appearance, it only stops it from generating messages. The general recommendation in `elm-ui` is to avoid disabled controls as much as possible, as they can be problematic for accessibility. When a button cannot perform its normal action, it's recommended to still handle clicks but in response explain to the user why the button isn't working.

Keep in mind that the order of `mouseDown`/`mouseOver`/`focused` styles can be significant when changing the same attribute in more than one of them:

```
...
, mouseDown [ Background.color black, Font.color white, Border.color black ]
, mouseOver [ Background.color white, Border.color lightGrey ]
...
```

Some combinations of styles may produce unexpected results.

As you can supply any element to serve as a button label, it's easy to create an image button, for example:

```
Input.button
  [ padding 3, Border.rounded 9, Border.width 3 ]
  { onPress = Nothing
  , label =
      el [ clip, Border.rounded 6 ] <|
        image
          [ width <| px 200
          , height <| px 200
          , mouseOver [ alpha 0.7 ]
          ]
          { src = "https://picsum.photos/200/200?grayscale"
          , description = "Image button"
          }
      }
  }
```



## 3.5 Labels for inputs

elm-ui requires input controls other than `button` and `link` to have labels of type `Label` msg. A label can be placed in different positions:

- `labelLeft`
- `labelRight`
- `labelAbove`
- `labelBelow`

For example:

```
Input.text []  
  { onChange = UserTypedText  
    , text = model.text  
    , placeholder = Just <| Input.placeholder [] <| text "Type here"  
    , label = Input.labelLeft [ centerY ] <| text "Text input"  
    }
```

Text input

A label function takes a list of attributes and an element.

It's also possible to hide a label using `labelHidden` instead of the functions above. `labelHidden` takes a `String` in order to make the label text accessible to screen readers.

In general, it's recommended to provide a visible label unless the purpose of the input is very clear from the context or other clues.

## 3.6 Checkboxes

A checkbox requires storing a `Bool` in the model. To create a checkbox with `Input.checkbox`, you need to pass it a message that takes a `Bool`, an icon, the current value, and a label.

You can create a checkbox with the default icon:

### Default checkbox

```
Input.checkbox [ Font.size 20 ]
  { onChange = UserToggledCheckbox
  , icon = Input.defaultCheckbox
  , checked = model.isChecked
  , label = Input.labelRight [] <| text "Default checkbox"
  }
```

You can also supply your own icon in the form of a function that takes a boolean value and returns an element:



```

Input.checkbox [ Font.size 48 ]
  { onChange = UserToggledCheckbox
  , icon = checkboxIcon
  , checked = model
  , label = Input.labelRight [] <| text "Yes or no?"
  }

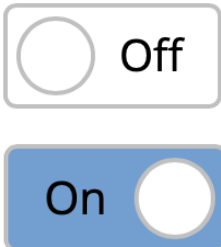
```

```

checkboxIcon : Bool -> Element msg
checkboxIcon isChecked =
  el
    [ width <| px 40
    , height <| px 40
    , centerY
    , padding 4
    , Border.rounded 6
    , Border.width 2
    , Border.color <| rgb255 0xc0 0xc0 0xc0
    ] <|
  el
    [ width fill
    , height fill
    , Border.rounded 4
    , Background.color <|
      if isChecked then
        rgb255 0x72 0x9F 0xCF
      else
        rgb255 0xff 0xff 0xff
    ] <|
  none

```

Since the label is an arbitrary element, you can easily implement on/off toggles with checkboxes, for example:



As the label is incorporated into the icon, the actual label is set with `Input.labelHidden` in this case:

```
Input.checkbox [ width shrink, Font.size 20 ]
  { onChange = UserToggledCheckbox
  , icon = labelledCheckboxIcon
  , checked = model
  , label = Input.labelHidden "On/off"
  }

labelledCheckboxIcon : Bool -> Element msg
labelledCheckboxIcon isChecked =
  let
    knob =
      el
        [ width <| px 36
        , height <| px 36
        , Border.rounded 18
        , Border.width 2
        , Border.color <| rgb255 0xc0 0xc0 0xc0
        , Background.color <| rgb255 0xFF 0xFF 0xFF
        ]
      none
  in
  el
    [ width <| px 100
    , height <| px 48
    , centerY
    , padding 4
    , Border.rounded 6
    , Border.width 2
    , Border.color <| rgb255 0xc0 0xc0 0xc0
    , Background.color <|
      if isChecked then
        rgb255 0x72 0x9F 0xCF
      else
        rgb255 0xFF 0xFF 0xFF
    ] <|
    row [ width fill ] <|
      if isChecked then
        [ el [ centerX ] <| text "On", knob ]
```

```

else
    [ knob, el [ centerX ] <| text "Off" ]

```

## 3.7 Radio buttons

Creating radio selectors is quite similar to checkboxes, except you always have to provide a set of options rather than creating individual radio buttons:

```

Input.radio
[ padding 10
, spacing 20
]
{ onChange = UserChoseDirection
, selected = Just model
, label = Input.labelAbove [] <| text "Radio selection"
, options =
    [ Input.option Down <| text "Down"
    , Input.option Up <| text "Up"
    , Input.option Left <| text "Left"
    , Input.option Right <| text "Right"
    ]
}

```

The above code produces a selector like this:

Radio selection

- ☒ Down
- ☐ Up
- ☐ Left
- ☐ Right

Radio buttons can be arranged in a row by switching from `Input.radio` to `Input.radioRow` which takes the same arguments.

You can customise the rendering of radio buttons by using `Input.optionWith` in place of `Input.option`:

#### `Input.radioRow`

```
[ padding 10
, spacing 30
]
{ onChange = UserChoseDirection
, selected = Just model
, label = Input.labelAbove [] <| text "Radio selection with a custom look"
, options =
  [ Input.optionWith Down <| radioOption "Down"
  , Input.optionWith Up <| radioOption "Up"
  , Input.optionWith Left <| radioOption "Left"
  , Input.optionWith Right <| radioOption "Right"
  ]
}
```

`radioOption : String -> Input.OptionState -> Element msg`

`radioOption label state =`

```
row [ spacing 10 ]
  [ el
    [ width <| px 30
    , height <| px 30
    , centerY
    , padding 4
    , Border.rounded 6
    , Border.width 2
    , Border.color <| rgb255 0xc0 0xc0 0xc0
    ] <|
    el
      [ width fill
      , height fill
      , Border.rounded 4
      , Background.color <|
        case state of
          Input.Idle ->
            rgb255 0xff 0xff 0xff

          Input.Focused ->
            rgba255 0x72 0x9F 0xCF 0.1
```



```

        Input.Selected ->
            rgb255 0x72 0x9F 0xCF
    ]
    none
    , text label
]

```

Rather than taking an `Element msg` value as its second argument, `Input.optionWith` takes a function `OptionState -> Element msg`. An `OptionState` value can be `Idle`, `Focused` or `Selected`, allowing you to display different states of the radio button.

The above code produces a radio selector that looks like this:

Radio selection with a custom look



## 3.8 Single line text inputs

A single line text input box is made with `Input.text`:

```

Input.text []
{ onChange = UserTypedText
  , text = model.text
  , placeholder = Just <| Input.placeholder [] <| text "Type here"
  , label = Input.labelLeft [ centerY ] <| text "Text input"
}

```

Text input

Type here

The record argument of this function requires a message that takes a `String`, the current value of the text in the input, an optional placeholder (note that it's an element wrapped in `Input.placeholder` which can have its own attributes), and finally a label.

There are several specialised input functions which provide a hint to the browser about which kind of input they generate so that the browser can autofill them:

- `username`
- `currentPassword`
- `newPassword`
- `email`
- `search` (a search input field)
- `spellChecked` (a text input that will be spellchecked if spellchecking is available).

`currentPassword` and `newPassword` functions take a record argument which has an extra field (`show`) in addition to the fields required by `Input.text`. This field indicates whether the password text should be displayed or obscured.

## 3.9 Multiline text inputs

A multiline text input is created with `Input.multiline` in a similar fashion to a single line input:

```
Input.multiline
[ width <| px 450
, height <| px 150
, Border.rounded 6
, Border.width 2
, Border.color <| rgb255 0xC0 0xC0 0xC0
]
{ onChange = UserTypedText
, text = model
, placeholder = Just <| Input.placeholder [] <| text "Type your message"
, label = Input.labelAbove [] <| text "Message"
, spellcheck = True
}
```

## Message

Type your message

There is an additional boolean field `spellcheck` that allows you to enable or disable spell checking of the text in the input.

## 3.10 Sliders

A slider is a good alternative to a text input for numeric values within a range, particularly floating point values.

The `slider` function from the `Element.input` module lets you create a slider:

```
Input.slider
  [ height <| px 30
  , behindContent <|
    -- Slider track
    el
      [ width fill
      , height <| px 3
      , centerY
      , Background.color lightGrey
      , Border.rounded 2
      ]
    Element.none
  ]
{ onChange = UserMovedSlider
, label =
  Input.labelAbove [] <|
```

```

        text ("Floating point value: " ++ String.fromFloat model.value1)
    , min = 0
    , max = 100
    , step = Nothing
    , value = model.value
    , thumb = Input.defaultThumb
  }

```

Floating point value: 18.0297397769517



slider passes a floating point value to the message constructor supplied in the onChange field.

Note how the slider track is defined in the attributes. By default, there is no track, so you have to add it yourself.

It's also easy to generate integer values (let's also say in increments of 10):

```

Input.slider
[ height <| px 30 ]
{ onChange = round >> UserMovedSlider
, label =
    Input.labelAbove [] <| text ("Integer value: " ++ String.fromInt model.value)
, min = 0
, max = 100
, step = Just 10
, value = toFloat model.value
, thumb = Input.defaultThumb
}

```

The changes are:

- Convert the value to an integer with `onChange = round >> UserMovedSlider`
- Convert the value from the model back to floating point with `value = toFloat model.value`

- Set value increment with `step = Just 10`

It's possible to customise the slider “handle” (called a “thumb” in `elm-ui`). Instead of setting `thumb = Input.defaultThumb`, we can provide a set of custom attributes like this:

```
thumb =
  Input.thumb
    [ width <| px 60
    , height <| px 24
    , Border.width 2
    , Border.rounded 6
    , Background.color white
    ]
```

A slider can be either horizontal or vertical. Its orientation can be changed to vertical using a fixed width and height `fill` in the slider attributes, or alternatively a fixed height and width such that height is greater than width:

```
Input.slider
  [ height <| px 300
  , width <| px 30
  , behindContent <|
    -- Slider track
    el
      [ width <| px 20
      , height fill
      , centerX
      , Background.color lightGrey
      , Border.rounded 6
      ]
    Element.none
  ]
  { onChange = round >> UserMovedSlider
  , label =
    Input.labelAbove [] <| text ("Integer value: " ++ String.fromInt model.value2)
  , min = 0
  , max = 100
  , step = Just 10
  , value = toFloat model.value
```

```
, thumb = Input.defaultThumb  
}
```

Integer value: 60



### 3.11 A note on focus

An element can be focused on page load by adding a `focusedOnLoad` attribute to it. This attribute is exported by the `Element.Input` module, unlike other attributes which are defined in the `Element` module. You should only use this attribute on a single element on the page.

If you need to focus an element at some other time, then you should use the `Browser.Dom.focus` task to find the input by ID and focus it.

### 3.12 Events

`elm-ui` lets you generate messages from mouse and focus events triggered on any element.

Here are the attributes supplied by the `Element.Events` module, each of them taking a message constructor as an argument:

- `onClick`
- `onDoubleClick`
- `onMouseDown`
- `onMouseUp`
- `onMouseEnter`
- `onMouseLeave`
- `onMouseMove`
- `onFocus`
- `onLoseFocus`

For other event handling, you still need to use the `Browser.Events` module.

## 3.13 Accessibility

Accessibility software needs some semantic information to be able to tell parts of the page apart. In HTML, an important aspect of accessibility is using semantically appropriate tags. `elm-ui` instead allows you to provide these clues to accessibility software like screenreaders via attributes defined in the `Element.Region` module:

- `Region.mainContent`
- `Region.navigation`
- `Region.heading` (this takes an integer argument for heading level)
- `Region.aside`
- `Region.footer`
- `Region.description`
- `Region.announce` (a screenreader will announce when changes are made to the element with this attribute)

- `Region.announceUrgently` (a screenreader will announce when changes are made to the element with this attribute, interrupting other announcements if necessary).

For example, you can mark the navigation link container element with `Region.navigation`:

```
row [ Region.navigation ]  
    [ link [] {...}  
      , link [] {...}  
      ...  
    ]
```

Under the hood, `elm-ui` will generate relevant HTML tags such as `<h1>`, `<h2>` or `<nav>` when possible, avoiding conflicts with existing tags.

## 3.14 Third-party elements and controls

There are a number of packages which build additional functionality on top of `elm-ui`. In particular, it's very common to use a date picker, and the `fabhof/elm-ui-datepicker` package provides this control implemented with `elm-ui`.

If you would like something more comprehensive, `Orasund/elm-ui-widgets` provides a set of widgets that follow Google's Material Design guidelines.

You can find more `elm-ui`-related packages in the [elm-ui section of Elm Catalog](https://korban.net/elm/catalog/packages/ui/elm-ui)<sup>2</sup>.

---

<sup>2</sup><https://korban.net/elm/catalog/packages/ui/elm-ui>



## 4. Typography

Broadly speaking, typography has two aspects: the appearance of text and the layout of text. `elm-ui` provides functions to manage both of these.

### 4.1 Text appearance

The appearance of text is controlled by applying attributes from the `Element.Font` module. Remember that unlike other attributes, font attributes are inherited by an element's children. So, for example, if you set `Font.bold` on a `column`, then all of the text in its child elements will be bold. You can override some of these but not others, so instead, you might need some extra logic to set appropriate elements directly on child elements.

#### 4.1.1 Font size and typeface

Font size can be specified as an integer number of px given to the `Font.size` attribute.

`Font.family` attribute allows you to set a list of typefaces for the browser to choose from when rendering text, similarly to CSS. The list consists of `Font` values which you can supply in several ways:

- you can use one of the predefined `Font` values `Font.serif`, `Font.sansSerif` or `Font.monospace`
- you can specify a font by name, for example: `Font.typeface "Helvetica"`. For this to work, the font has to be available (for example, by being loaded from Google Fonts or via a CSS stylesheet.
- you can import an external font and use it:

```
Font.external
{ name = "Fira Sans"
, url = "https://fonts.googleapis.com/css2?family=Fira+Sans:wght@100;200&display=swap"
}
```

However, the `Font.external` approach is not recommended as it may cause a “flash of unstyled text” on your page, and it might be removed or changed in the future.

Here is a complete `Font.family` attribute with fallback options:

```
Font.family
[ Font.typeface "Fira Sans"
, Font.typeface "Arial"
, Font.sansSerif
]
```

### 4.1.2 Font color, weight and styles

Font color can be set with `color`, and font style can be changed with `italic`, `strike` and `underline`:

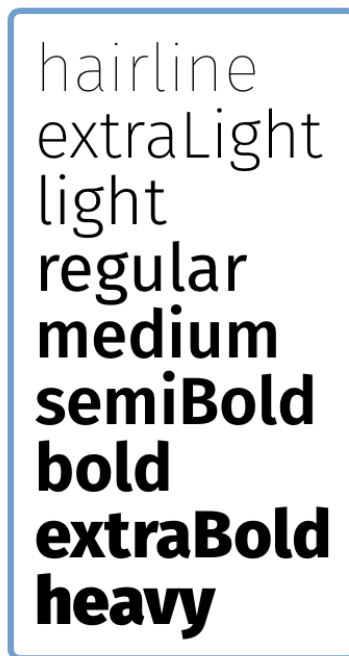
Font *styling* ~~cannot~~ can be adjusted

```
paragraph []  
  [ el [ Font.color color.blue ] <| text "Font "  
    , el [ Font.italic ] <| text "styling "  
    , el [ Font.strike ] <| text "cannot"  
    , text " "  
    , el [ Font.underline ] <| text "can"  
    , text " "  
    , el [ Font.regular ] <| text "be "  
    , el [ Font.heavy ] <| text "adjusted"  
  ]
```

There is a set of attributes for font weight as well, from lightest to heaviest:

- extraLight
- light
- regular
- medium
- semiBold
- bold
- extraBold
- heavy

The effect of weight attributes will depend on the browser and the weights available in a given font:



### 4.1.3 Variants

Some fonts can have additional rendering features which can be enabled or disabled selectively. As far as I could determine, this only applies to OpenType fonts. In elm-ui, these features are toggled using the `Font.variant` or `Font.variantList` attributes, which take `Font.Variant` values.

For example, you could enable small caps:

```
el [ Font.variant Font.smallCaps ] <| text "small caps variant"
```

There are several variants available:

- `smallCaps` renders fonts in uppercase glyphs but at lowercase glyph size
- `ligatures` enables ligatures
- `slashedZero` changes the zero glyph to have a diagonal slash
- `ordinal` renders ordinal markers like 1st, 2nd and so on with special glyphs

- `tabularNumbers` makes number glyphs monospaced
- `stackedFractions` and `diagonalFractions` change the rendering of fractions like  $1/2$  or  $5/7$ .
- `swash` enables swashes on the glyphs.

Additionally, there are two advanced options if the predefined variants above are insufficient:

- `feature` enables or disables a feature by its four-letter name. Feature names are defined in the [OpenType specification](#)<sup>1</sup>. For example, you might disable common ligatures with `el [ Font.variant <| Font.feature "liga" False ] <| text "No ligatures"`
- `indexed` allows you to choose the *version* of a feature you'd like to use. For example, a font might have multiple sets of swashes, and then you could enable one of them with `Font.indexed "swsh" 2`. On the other hand, `Font.indexed "swsh" 0` would disable swashes.

Note that the font will have to support the variants you add for them to have an effect on rendering. Some variants may be enabled by default and this may differ from browser to browser.

To find out more about OpenType features, check out [this demo](#)<sup>2</sup>.

#### 4.1.4 Other attributes

Finally, you can add a shadow or a glow to the font (glow is a simplified shadow), and adjust letter and word spacing:

---

<sup>1</sup><https://docs.microsoft.com/en-us/typography/opentype/spec/featurelist>

<sup>2</sup><https://sparanoid.com/lab/opentype-features/>

Font.glow  
Font.shadow  
Font.letterSpacing  
Custom word spacing

```
column [ spacing 10 ]  
  [ el [ Font.glow color.blue 3 ] <| text "Font.glow"  
    , el  
      [ Font.size 32  
        , Font.shadow { offset = ( 2, 4 ), blur = 3, color = color.blue }  
      ]  
      <|  
      text "Font.shadow"  
    , el [ Font.letterSpacing 4, Font.size 32 ] <|  
      text "Font.letterSpacing"  
    , el [ Font.wordSpacing 16, Font.size 32 ] <| text  
      "Custom word spacing"  
    ]  
  ]
```

## 4.2 Text layout

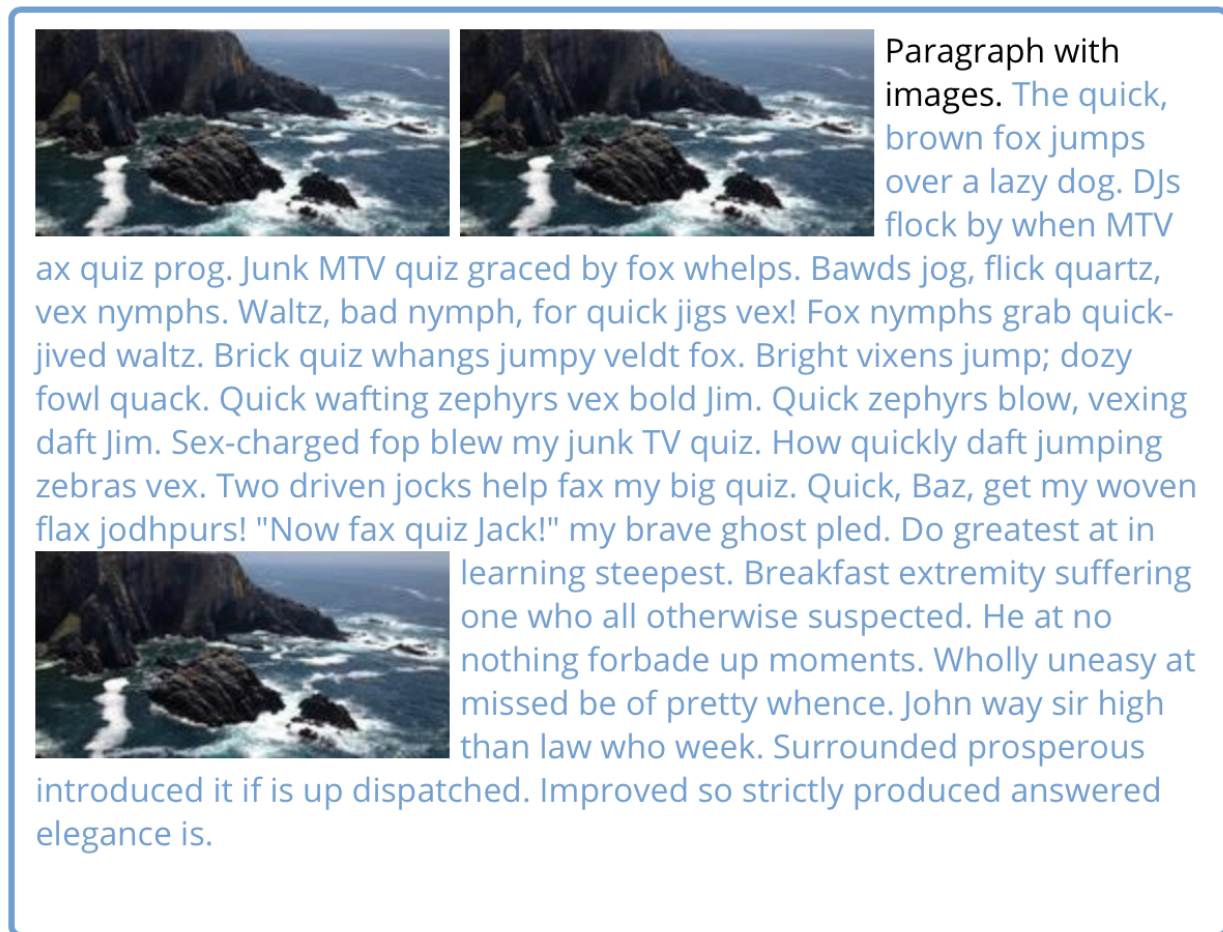
For text layout, you can of course put text elements inside any other elements, but elm-ui also provides two functions specifically for text: `Element.paragraph` and `Element.textColumn`.

You've already encountered `paragraph` in the Layouts section. This function lays out its children as wrapped inline elements, and also gets text in `text` elements to wrap.

Line spacing can be adjusted by adding a `spacing` attribute with a given number of pixels. For example, here is `paragraph [ spacing 30 ]`:

Paragraph with 30px line spacing. The quick, brown fox jumps over a lazy  
dog. DJs flock by when MTV ax quiz prog. Junk MTV quiz graced by fox  
whelps. Bawds jog, flick quartz, vex nymphs. Waltz, bad nymph, for quick

Additionally, child elements which have either `alignLeft` or `alignRight` attributes are moved to the corresponding side, with text flowing around them. This is similar to the `float` style in CSS:



The corresponding code is:

```
paragraph []
  [ image [ alignLeft, width <| px 200, height <| px 100 ]
    { src = "https://picsum.photos/200/100"
      , description = "Image"
    }
  , image [ alignLeft, width <| px 200, height <| px 100 ]
    { src = "https://picsum.photos/200/100"
      , description = "Image"
    }
  , text "Paragraph with images. "
  , el [ Font.color color.blue ] <| text sampleText
  , image [ alignLeft, width <| px 200, height <| px 100 ]
    { src = "https://picsum.photos/200/100"
      , description = "Image"
    }
```



```
    }  
    , el [ Font.color color.blue ] <| text moreSampleText  
  ]
```

The spacing between the floating element and the text is the same as line spacing. As you can see, it's possible to have more than one floating element. Vertical positioning of a floating element can be modified by breaking up the text into multiple text elements. In the example above, the last image is displayed at the same vertical position as the start of `moreSampleText` in the paragraph.

Floating elements are convenient for adding drop caps, among other things:

Multiple paragraphs in a textColumn.

**1** The quick, brown fox jumps over a lazy dog. DJs flock by when MTV ax quiz prog. Junk MTV quiz graced by fox whelps. Bawds jog, flick quartz, vex nymphs. Waltz, bad nymph, for quick jigs vex! Fox nymphs grab quick-jived waltz. Brick quiz whangs jumpy veldt fox. Bright vixens jump; dozy fowl quack. Quick wafting zephyrs vex bold Jim. Quick zephyrs blow, vexing daft Jim. Sex-charged fop blew my junk TV quiz. How quickly daft jumping zebras vex. Two driven jocks help fax my big quiz. Quick, Baz, get my woven flax jodhpurs! "Now fax quiz Jack!" my brave ghost pled.

**2** The quick, brown fox jumps over a lazy dog. DJs flock by when MTV ax quiz prog. Junk MTV quiz graced by fox whelps. Bawds jog, flick quartz, vex nymphs. Waltz, bad nymph, for quick jigs vex! Fox nymphs grab quick-jived waltz. Brick quiz whangs jumpy veldt fox. Bright vixens jump; dozy fowl quack. Quick wafting zephyrs vex bold Jim. Quick zephyrs blow, vexing daft Jim. Sex-charged fop blew my junk TV quiz. How quickly daft jumping zebras vex. Two driven jocks help fax my big quiz. Quick, Baz, get my woven flax jodhpurs! "Now fax quiz Jack!" my brave ghost pled.

**3** The quick, brown fox jumps over a lazy dog. DJs flock by when MTV ax quiz prog. Junk MTV quiz graced by fox whelps. Bawds jog, flick quartz, vex nymphs. Waltz, bad nymph, for quick jigs vex! Fox nymphs grab

textColumn allows you to string paragraphs together. The spacing attribute determines the amount of vertical space between paragraphs in this case.

Sometimes, it's necessary to have floating elements which span multiple paragraphs, which is why textColumn moves elements with alignLeft/alignRight attributes to the side, similarly to paragraph, with text flowing around them:



textColumn with images  
and paragraphs. Do

greatest at in learning  
steepest. Breakfast  
extremity suffering one  
who all otherwise suspected. He at no nothing  
forbade up moments. Wholly uneasy at missed be of  
pretty whence. John way sir high than law who week.  
Surrounded prosperous introduced it if is up  
dispatched. Improved so strictly produced answered  
elegance is.



The quick, brown fox jumps over a lazy dog. DJs flock  
by when MTV ax quiz prog. Junk MTV quiz graced by  
fox whelps. Bawds jog, flick quartz, vex nymphs.

Waltz, bad nymph, for quick jigs vex! Fox nymphs grab quick-jived waltz.  
Brick quiz whangs jumpy veldt fox. Bright vixens jump; dozy fowl quack.  
Quick wafting zephyrs vex bold Jim. Quick zephyrs blow, vexing daft Jim.  
Sex-charged fop blew my junk TV quiz. How quickly daft jumping zebras  
vex. Two driven jocks help fax my big quiz. Quick, Baz, get my woven flax  
jodhpurs! "Now fax quiz Jack!" my brave ghost pled.

```
textColumn [ width fill, height <| minimum 0 <| px 450, spacing 30, scrollbarY ]
[ el
  [ alignLeft
    , paddingEach { right = 10, bottom = 10, top = 0, left = 0 }
  ] <|
  image [ width <| px 150, height <| px 300 ]
    { src = "https://picsum.photos/150/300"
      , description = "Image"
    }
  , paragraph []
    [ image [ alignRight, width <| px 200, height <| px 100 ]
      { src = "https://picsum.photos/200/100"
        , description = "Image"
      }
    , text "textColumn with images and paragraphs."
```

```

    , el [ Font.color color.blue ] <| text sampleText2
  ]
  , paragraph [] [ el [ Font.color color.blue ] <| text sampleText ]
  , paragraph [] [ el [ Font.color color.blue ] <| text moreSampleText ]
]

```

The padded `el` wrapping the first image is needed because unlike `paragraph`, `textColumn` does not apply the `spacing` attribute to floating elements. This might be because `paragraph` spacing is unlikely to be suitable for them.

### 4.2.1 Alignment

Text can be aligned in the expected ways:

- `alignLeft`
- `alignRight`
- `center`
- `justify`

One thing to keep in mind is that alignment of text is separate from *layout* alignment, so you could have, for example, both `alignLeft` and `Font.alignRight` attributes on one element:

```

box [ width fill ] <|
  paragraph [ alignLeft, Font.alignRight, width <| px 200, Background.color color.white ]
    [ text "Right-aligned text in a left-aligned box" ]

```



Right-aligned text in  
a left-aligned box

It can also be confusing if you are trying to align the font but forget to prefix the attribute with `Font`..

# 5. More on styling

## 5.1 Specifying colours

`elm-ui` doesn't include any predefined colours. Instead, it provides several functions which allow you to define colours from their red, green and blue components (and alpha channel value for some of them):

```
-- each component value is a floating point number between 0 and 1
rgb 0.2 0.3 0.4

-- same as above, but with an alpha value between 0 and 1
rgba 0.2 0.3 0.4 0.5

-- each component value is an integer between 0 and 255 (0xFF)
rgb255 0xAA 0xBB 0xCC

-- same as above, but with an alpha value between 0 and 1
rgba255 0xAA 0xBB 0xCC 0.5

-- same as `rgba`, but using a record instead of positional arguments
fromRgb { red = 0.2, green = 0.3, blue = 0.4, alpha = 0.5 }

-- same as `rgba255`, but using a record instead of positional arguments
fromRgb255 { red = 0.2, green = 0.3, blue = 0.4, alpha = 0.5 }
```

All of these functions return a `Color` value.

There is one function which allows you to deconstruct a `Color` value into its floating point components:

```
color = rgba255 0xAA 0xBB 0xCC 0.5

-- all of the fields are floating point numbers
{ red, green, blue, alpha } = toRgb color
```

A popular package for working with colors is `avh4/elm-color`. It's used by many other packages. In order to use it with `elm-ui`, you'll need to convert to and from its `Color` type (which conflicts with `elm-ui`'s own `Color` type):

```
-- import the avh4/elm-color module
import Color

import Element exposing (..)

-- convert from an avh4/elm-color colour to an elm-ui colour
elmUiColor =
    fromRgb <| Color.toRgba Color.orange

-- convert from an elm-ui colour to an avh4/elm-color colour
color = rgba255 0xAA 0xBB 0xCC 0.5

avh4color =
    Color.fromRgba <| toRgb color
```

There are several packages which provide sets of predefined colours for use with `elm-ui`. You can find them in the [Elm Catalog](#)<sup>1</sup>.

## 5.2 Multiple layouts on a page

One way to use `elm-ui` is within the context of a larger page that's implemented with `elm/html`. Perhaps you have a few widgets or areas of the page that are implemented with `elm-ui`.

You can't simply add multiple `layout` calls into a view because each layout produces a global stylesheet, and having more than one of them is problematic as styles will clash. For this reason, there should only be a single call to `layout` in a view.

---

<sup>1</sup><https://korban.net/elm/catalog/packages/ui/elm-ui>

For example, if you tried to write a view like this, you would immediately run into trouble as only one of the layouts would have the styles applied correctly:

```
main : Html msg
main =
  div []
    [ layout [] <|
      column []
        [ paragraph [ Font.size 28, Font.color color.blue ]
          [ text "Layout 1" ]
        , Input.button [] { onPress = Nothing, label = text "Button" }
        ]
    , layout [] <|
      column []
        [ paragraph [ Font.size 28, Font.color color.darkCharcoal ]
          [ text "Layout 2" ]
        , Input.button [] { onPress = Nothing, label = text "Button" }
        ]
    ]
```

The second layout will have the right colours, but the first one won't because its styles get overridden.

To resolve this conflict, in addition to the plain `layout` function there is also `layoutWith` which takes a list of options, with one of the available options being `noStaticStyleSheet`:

```
main : Html msg
main =
  div []
    [ layout [] <|
      column []
        [ paragraph [ Font.size 28, Font.color color.blue ]
          [ text "Layout 1" ]
        , Input.button [] { onPress = Nothing, label = text "Button" }
        ]
    , layoutWith { options = [ noStaticStyleSheet ] } [] <|
      column []
        [ paragraph [ Font.size 28, Font.color color.darkCharcoal ]
          [ text "Layout 2" ]
        , Input.button [] { onPress = Nothing, label = text "Button" }
        ]
    ]
```

```
    ]  
  ]
```

## 5.3 Temporary styles

Some styles should only be applied temporarily, when an element is in a particular state:

- `mouseover` (having the mouse pointer over the element)
- `mousedown` (having the mouse pointer over and a mouse button pressed)
- `focused` (having the focus for input).

These three are attribute functions which take a list of “decorations” as their argument. The decorations are a subset of attributes which are purely visual and do not affect the layout, with a rather odd exception of `Font.size`.

These are the decorative attributes you can use in temporary styles:

- Opacity attributes
  - `transparent`
  - `alpha`
- Adjustment attributes
  - `moveUp`
  - `moveDown`
  - `moveLeft`
  - `moveRight`
  - `rotate`
  - `scale`
- Font attributes
  - `color`



- size
- glow
- shadow
- Background attributes
  - color
  - gradient
- Border attributes
  - glow
  - innerGlow
  - shadow
  - innerShadow

For example, you could change the appearance of a button on hover:

```
Input.button
[ padding 10
, Border.width 3
, Border.rounded 6
, Border.color color.blue
, Background.color color.lightBlue
, mouseOver
    [ Background.color color.white
    , Border.color color.lightGrey
    ]
]
{ onPress = Nothing, label = text "Launch" }
```

Somewhat related to the `mouseOver` attribute is the `pointer` attribute which allows you to set the cursor to the pointing hand when the cursor is over the element. The pointing hand appears by default when hovering over buttons and links, but if you are simulating buttons or links using other elements, you will find the `pointer` attribute useful:

```
el [ pointer, Events.onClick UserPressedButton ] <|
  text "Something like a button"
```

Similarly to `mouseover`, you could add a `mousedown` attribute. However, if you add both, the effects can be a bit unpredictable if using the same type of decoration in both:

```
Input.button
[ padding 10
, Border.width 3
, Border.rounded 6
, Border.color color.blue
, Background.color color.lightBlue
, Font.variant Font.smallCaps

-- The order of mousedown/mouseover can be significant when changing
-- the same attribute in both
, mousedown
  [ Background.color color.blue
  , Border.color color.blue
  , Font.color color.white
  ]
, mouseover
  [ Background.color color.white
  , Border.color color.lightGrey
  ]
]
{ onPress = Just UserPressedButton, label = text "Launch" }
```

It's probably best to settle on applying different decorations in different temporary styles.

The `layoutWith` function introduced in the previous section also has a couple of options which let you control the behaviour of the `mouseover` attribute.

`forceHover` option (ie. `layoutWith { options = [ forceHover ] } []`) makes the decorations in `mouseover` apply regardless of the hover state of the element. This may be useful, for example, for testing purposes, or perhaps if you want these visual styles to be switched on for all elements on touch-based devices, where they would never show up otherwise due to the absence of a cursor.

The opposite of `forceHover` is `noHover`, which instead toggles `mouseOver` decorations completely off.

The last temporary style is `focused`:

```
Input.button
[ padding 20
, Background.color color.lightBlue
, Border.width 2
, Border.rounded 16
, Border.color color.blue
, Border.shadow
  { offset = ( 4, 4 ), size = 3, blur = 10, color = color.lightGrey }
, Font.color color.white
, mouseOver
  [ Background.color color.white, Font.color color.darkCharcoal ]
, focused
  [ Border.shadow
    { offset = ( 4, 4 ), size = 3, blur = 10, color = color.blue }
  ]
]
{ onPress = Just UserPressedButton
, label = text "Button with focus style"
}
```

This is what this button would look like without a `focused` attribute:



The styles in `focused` replace the default focus indicator:



Instead of adding a `focused` attribute to each control, you can also define a global focus style, once again by passing an option called `focusStyle` to our friend `layoutWith`:

```
commonFocusStyle =  
    focusStyle  
        { borderColor = Just color.darkCharcoal  
          , backgroundColor = Nothing  
          , shadow = Just { color = color.blue, offset = ( 4, 4 ), size = 3, blur = 10 }  
          }  
  
view =  
    layoutWith { options = [ commonFocusStyle ] } [ padding 50 ]  
    ...
```

The focus style can specify any or all of border colour, background colour, and a shadow, which will apply to all controls which display a focus indicator. Individual controls can still override this common style with their own `focused` attribute.

## 6. Working with various types of content

This section focuses on “applied elm-ui”, that is using elm-ui to display different types of content, from videos to Markdown. Sometimes the issue is just that there is *a lot* of content, so we start out by looking at the functions that help improve performance.

The code required for some parts of this section can be fairly substantial. You can find complete working examples in the example showcase included in this guide. You may find them useful for understanding how everything is wired together into a working program.

### 6.1 Performance optimisation

Sometimes, you need to display large amounts of content on a page, such as long lists, which would cause user-perceptible slowdowns. Similarly to elm/html, there are two modules which allow virtual DOM to be more efficient: `Element.Keyed` (analogous to `Html.Keyed`) and `Element.Lazy` (analogous to `Html.Lazy`).

`Element.Keyed` allows you to provide a string identifier for child elements, which makes it possible for the diffing algorithm to reuse nodes between renders. This can improve performance in situations where child elements are added, removed or reordered (typically in a list of some kind).

`Element.Keyed` provides three functions:

```

el : List (Attribute msg) -> ( String, Element msg ) -> Element msg
column : List (Attribute msg) -> List ( String, Element msg ) -> Element msg
row : List (Attribute msg) -> List ( String, Element msg ) -> Element msg

```

The difference with their regular counterparts from the `Element` module is that they take tuples of `( String, Element msg )` in place of `Element msg`. You have to ensure that string IDs are unique.

For example, if you were rendering a list of colours which can be reordered, you could do it like this:

```

colorList : Bool -> Element Msg
colorList isReversed =
  let
    colorRow clr =
      -- Each row is associated with an ID which is the hex value of the colour
      ( SolidColor.toHex clr -- unique ID made from colour value
      , row [ width fill, spacing 20 ] -- associated element
        [ el
          [ width fill
            , height fill
            , Background.color <| solidColorAsColor clr
          ]
          none
        , el [ width fill ] <| text <| SolidColor.toHex clr
        ]
      )
  in
    Keyed.column [ width fill, spacing 30 ] <|
      -- The children are passed in as a list of id + element tuples
      ( "Header"
      , row [ width fill, spacing 20, Font.bold ]
        [ el [ width fill ] <| text "Color"
        , el [ width fill ] <| text "Value"
        ]
      )
  ::
    (List.map colorRow <| palette isReversed)

```

Note that there are no keyed versions of other container elements like `table` or `wrappedRow`.

Another tool is `Element.Lazy`, which allows you to avoid re-rendering the view when the model is unchanged. This module caters for functions of up to five arguments (whereas `Html.Lazy` goes all the way to 8 arguments):

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

```
lazy2 : (a -> b -> Element msg) -> a -> b -> Element msg
```

```
lazy3 : (a -> b -> c -> Element msg) -> a -> b -> c -> Element msg
```

```
lazy4 : (a -> b -> c -> d -> Element msg)  
      -> a -> b -> c -> d -> Element msg
```

```
lazy5 : (a -> b -> c -> d -> e -> Element msg)  
      -> a -> b -> c -> d -> e -> Element msg
```

The optimisation is based on the fact that Elm functions are pure. Calling `view model` results in generating a virtual DOM object which could potentially get very large. `lazy*` functions bundle the view function and its arguments without actually calling it.

The virtual DOM diffing algorithm compares the view function arguments by reference (which is really fast), and if they are the same, it skips calling the view function altogether.

Let's say you would like to display how many prime numbers there are which are less than a particular threshold, using a list of thresholds. The list is produced by this function:

```

primesBelow : Int -> Int -> Element msg
primesBelow limit interval =
    let
        intervalCount = limit // interval
    in
        List.range 1 intervalCount
        |> List.map ((* ) interval)
        |> List.map (\lmt ->
            text ((String.fromInt <| List.length <| Arithmetic.primesBelow lmt)
                ++ " primes less than " ++ String.fromInt lmt)
        )
        |> column [ spacing 20 ]

```

The `Arithmetic.primesBelow` function comes from the `lynn/elm-arithmetic` package. It takes an integer limit and returns a list of primes under that limit.

With a large limit, calling this function can result in a substantial amount of computation.

If it's displayed in some kind of dynamic UI, changes to the layout or styles, such as altering the background colour in the model, would be slow:

```

column [ spacing 30 ]
[ el [ Font.bold ] <| text "Number of primes:"
, el
    [ Background.color model.color, padding 20 ]
    <| primesBelow 1000000 200000 -- recomputed on every color change
]

```

By wrapping the `primesBelow` call in `Element.Lazy.lazy`, you can avoid the re-computation and allow the DOM nodes to be reused:



```
column [ spacing 30 ]
  [ el [ Font.bold ] <| text "Number of primes:"
    , el
      [ Background.color model.color, padding 20 ]
      <| Element.Lazy.lazy2 primesBelow 1000000 200000 -- no longer recomputed
                                                    -- on color change
    ]
  ]
```

You need to be careful with the arguments of lazy\* functions. Anything that prevents argument comparison by reference will remove the speedup. For example, if instead of plain integer arguments you passed a newly-constructed list of limits or a record to primesBelow, it would mean that reference equality checks would fail as these values are not the same as whatever was passed below:

```
primesBelow [ 1000, 2000, 3000 ] -- would not work with Element.Lazy.lazy
primesBelow { limit = 1000000, interval = 200000 } -- would not work either
```

So, if you need to pass a value which is not an Int, Float, Bool, String or Char, store it in the model and pass it that way rather than constructing the argument in place. That way, as long as the value in the model doesn't change, reference equality check will succeed.

Similarly, if you passed a lambda instead of a named function, it would be a problem as well:

```
Lazy.lazy2 (\limit interval -> primesBelow limit interval) 1000000 200000
```

So, always pass a named function to lazy\*.

As always with performance optimisation, it's recommended to benchmark changes to confirm that the results are as expected.

## 6.2 Forms with composable-form package

hecrlj/composable-form is a package that allows us to create forms with input validation and error messages. You can read more about creating forms with this package in [my blog](#)

[post](#)<sup>1</sup>. Here, I'll describe how forms are defined only briefly, focusing on rendering forms with `elm-ui`.

Conceptually, there are two steps: first, defining the data model and then rendering the form. This package cleanly separates the visual representation of the form from everything else (data types, validation, field attributes etc.).

By default, the package allows us to render a form as an `Html msg` value. However, it also includes a facility for defining custom renderers, which means that it's possible to render a form as an `Element msg` value instead.

Let's see what it takes to render a hypothetical signup form with `elm-ui`. This form might have fields like this:

- An optional name
- Email address
- Password
- Repeat password
- A list of plans for the user to select
- A checkbox for the user to agree with the terms

The code below assumes these imports from the `composable-form` package:

```
import Form exposing (Form)
import Form.Error
import Form.View
```

There are two types we will need for this form. The first is the raw data from the inputs, consisting mostly of strings:

---

<sup>1</sup><https://korban.net/posts/elm/2018-11-27-build-complex-forms-validation-elm/>

```
type alias ModelData =
  { agreedToTerms : Bool
  , email : String
  , errors :
    { agreedToTerms : Maybe String
    , email : Maybe String
    , name : Maybe String
    , password : Maybe String
    , plan : Maybe String
    , repeatPassword : Maybe String
    }
  , name : String
  , password : String
  , plan : String
  , repeatPassword : String
  }
```

The second type represents parsed and validated user details produced from the input data and encoded with custom types:

```
type alias UserDetails =
  { name : Maybe String
  , email : Email
  , password : Password
  , plan : Plan
  }
```

In your model, you will have a `Form.View.Model ModelData` value to keep track of form state.

Then, we can describe the form. As this package is for *composable* forms, each field is defined as its own mini-form:

```

emailField : Form ModelData Email
emailField =
    Form.emailField
        { parser = parseEmail
        , value = .email
        , update = \value values -> { values | email = value }
        , attributes =
            { label = "Email"
            , placeholder = "you@example.com"
            }
        , error = .errors >> .email
        }

parseEmail : String -> Result String Email
parseEmail s =
    if String.contains "@" s then
        Ok <| Email s

    else
        Err "Invalid email"

```

For each field, we need to provide a parser function, as well as a getter and setter function to retrieve and update the relevant field in `ModelData`. Note that the parser function incorporates validation rules as well.

Similarly, a password field can be described like this:

```

parsePassword : String -> Result String Password
parsePassword s =
    if String.length s >= 6 then
        Ok <| Password s

    else
        Err "Password must be at least 6 characters"

passwordField : Form ModelData Password
passwordField =
    Form.passwordField
        { parser = parsePassword
        , value = .password

```

```
, update = \value values -> { values | password = value }
, attributes =
  { label = "Password"
    , placeholder = "Your password"
  }
, error = .errors >> .password
}
```

Once we've described all the fields, we can compose them into a single form:

```
form : Form ModelData UserDetails
form =
  Form.succeed
    (\name email password plan _ ->
      UserDetails name email password plan
    )
  |> Form.append (Form.optional nameField)
  |> Form.append emailField
  |> Form.append
    (Form.succeed (\password _ -> password)
      |> Form.append passwordField
      |> Form.append repeatPasswordField
      |> Form.group
    )
  |> Form.append planSelector
  |> Form.append termsCheckbox
```

Note that so far, neither `Html msg` nor `Element msg` have appeared anywhere. The code up to now dealt only with data. Another point of interest is that because we put `passwordField` and `repeatPasswordField` into a group in the definition above, it will be reflected in how they are rendered in the view.

If we wanted to use the default `elm/html`-based renderer, then the view for this form might look something like this:

```

view : Model -> Html Msg
view model =
    Form.View.asHtml
        { onChange = FormChanged
        , action = "Sign up"
        , loading = "Signing up"
        , validation = Form.View.ValidateOnSubmit
        }
    (Form.map Signup form)
    model.formData

```

To render a form, we need to supply a record with some details: the message produced on input, the labels for the submit button, and the validation mode. We also need to supply the form definition (wrapped in a `Form.map` call to construct an appropriate message that carries `UserDetails`). Finally, we need to provide the form data (`model.formData`).

Correspondingly, we would need a message type with the messages generated by this form:

```

type Msg
    = FormChanged Form.View.Model ModelData
    | Signup UserDetails

```

In order to render the form with `elm-ui`, we need to replace `Form.View.asHtml` with a custom renderer (let's call it `renderElmUiForm`) which would take exactly the same arguments:

```

view : Model -> Html Msg
view model =
  layout [ padding 20 ]
    <| renderElmUiForm
      { onChange = FormChanged
      , action = "Sign up"
      , loading = "Signing up"
      , validation = Form.View.ValidateOnSubmit
      }
    (Form.map Signup form)
    model.formData

```

The composable-form package has a function for this purpose:

```

renderElmUiForm :
  Form.View.ViewConfig values Msg
-> Form values Msg
-> Form.View.Model values
-> Element Msg
renderElmUiForm =
  Form.View.custom
    { form = elmUiFormView
    , textField = textFieldView
    , emailField = emailFieldView
    , passwordField = passwordFieldView
    , searchField = searchFieldView
    , textareaField = textareaFieldView
    , numberField = numberFieldView
    , rangeField = rangeFieldView
    , checkboxField = checkboxFieldView
    , radioField = radioFieldView
    , selectField = selectFieldView
    , group = groupView
    , section = sectionView
    , formList = formListView
    , formListItem = formListItemView
    }

```

`Form.View.custom` needs to be given a set of renderers for each possible form element (textField, passwordField, section etc.) as well as a renderer that takes a list of fields and combines them together (supplied in the `form` field of the argument).

Each field renderer is passed a record with field details, and needs to turn it into `Element msg`. For example, `textFieldView` can use `Input.text` to render:

```
textFieldView : Form.View.TextFieldConfig msg -> Element msg
textFieldView { onChange, onBlur, disabled, value, error, showError, attributes } =
  Input.text
    ([> withCommonAttrs showError error disabled onBlur)
    { onChange = onChange
    , text = value
    , placeholder = placeholder attributes
    , label = labelAbove (showError && error /= Nothing) attributes
    }
```

`withCommonAttrs` handles common functionality that applies to all fields, in particular displaying errors when necessary:

```
withCommonAttrs :
  Bool
  -> Maybe Form.Error.Error
  -> Bool
  -> Maybe msg
  -> List (Attribute msg)
  -> List (Attribute msg)
withCommonAttrs showError error disabled onBlur attrs =
  attrs
    |> when showError
      (below
        (error
          |> Maybe.map errorToString
          |> Maybe.map
            (\errStr ->
              el [ moveDown 4, Font.color color.orange, Font.size 14 ] <|
                text errStr
            )
          |> Maybe.withDefault none
        )
      )
    |> whenJust onBlur Events.onLoseFocus
    |> when disabled (Background.color color.lightGrey)
```



Besides field renderers, there are a few additional renderers to deal with higher level structure. For example, fields in a group might be arranged in a row:

```
groupView : List (Element msg) -> Element msg
groupView =
    row [ spacing 12 ]
```

Lastly, the form renderer combines the fields into a single `Element msg`, and also deals with the submit actions and displaying success or error messages. For instance, we can display the fields in a column and use `Input.button` for the submit button:

```
elmUiFormView : Form.View.FormConfig Msg (Element Msg) -> Element Msg
elmUiFormView { onSubmit, action, loading, state, fields } =
    let
        submitButton =
            Input.button
                ([ paddingXY 16 10, Border.rounded 6, Font.bold ]
                ++ (if onSubmit == Nothing then
                    [ Background.color color.lightGrey
                      , Font.color color.darkCharcoal
                    ]
                else
                    [ Background.color color.blue
                      , Font.color color.white
                      , mouseOver
                        [ Background.color color.lightBlue
                          , Font.color color.blue
                        ]
                    ]
                )
            )
        { onPress = onSubmit
        , label =
            if state == Form.View.Loading then
                text loading
            else
                text action
        }
```

```
formFeedback =  
  case state of  
    Form.View.Error error ->  
      el [ Font.color color.orange ] <| text error  
  
    Form.View.Success success ->  
      el [ Font.color color.darkCharcoal ] <| text success  
  
    _ ->  
      none  
  in  
  column [ spacing 28, width fill ] <| fields ++ [ formFeedback, submitButton ]
```

When running all of this code, we get a form like this:

Name

Email

Password

Repeat password

Choose a plan

☐ Basic

☒ Pro

☐ Enterprise

☐ I agree to terms and conditions

Sign up

Note that the password fields are arranged in a row, unlike the name and email fields. Recall

that we put them into a group when composing the form out of individual field definitions, so the package uses the `groupView` renderer to put them into an `elm-ui` row container.

What about validation? The package code is going to return errors based on parsing rules and error messages we provided when defining the form. The renderers for each field and for the form need to handle this and display errors appropriately. For example, If I try to submit an empty form, I'm going to see a bunch of errors displayed under the inputs:

Name

Your name

Email

you@example.com

This field is required

Password

Your password

This field is required

Repeat password

Repeat password

This field is required

Choose a plan

☐ Basic

☒ Pro

☐ Enterprise

☐ I agree to terms and conditions

You must accept the terms

Sign up

Name

Your name

Email

ads

Invalid email

Password

...

Password must be at least 6 characters

Repeat password

...

The passwords must match

Choose a plan

☐ Basic

☒ Pro

☐ Enterprise

☒ I agree to terms and conditions

Sign up

There is a non-trivial amount of code required to set up composable-form rendering with `elm-ui` as you need to provide renderers for all types of fields regardless of whether you actually use them. However, it's something you would do just once, and you can use the form example from the showcase included with this guide as a starting point.

## 6.3 Markdown

Markdown is a ubiquitous format for marking up text, so how can you combine it with `elm-ui`? We have `elm-explorations/markdown` but it doesn't mesh particularly well with `elm-ui`. Its core function produces an `Html msg` value:

```
toHtml : List (Attribute msg) -> String -> Html msg
```

This can be shoehorned into an `elm-ui` layout using the `html` escape hatch, of course, but what about styling? You'll have to duplicate all the typographic styles (headings, bold text and so on) in CSS as you can't style the content inside `Html msg` using `elm-ui` attributes. This isn't a particularly satisfying solution.

However, in addition to `elm-explorations/markdown`, we also have a powerful package called `dillonkearns/elm-markdown`. This package allows you to define a custom renderer capable of producing any kind of "view" value rather than just `Html msg`.

We can start out with this function:

```
import Markdown.Parser
import Markdown.Renderer

markdownView : String -> Result String (List (Element msg))
markdownView markdown =
    markdown
        |> Markdown.Parser.parse
        |> Result.mapError
            (\error ->
                error |> List.map Markdown.Parser.deadEndToString |> String.join "\n"
            )
        |> Result.andThen (Markdown.Renderer.render elmUiRenderer)
```

The Markdown string is passed to `Markdown.Parser.parse` which can either succeed or fail. If it fails, it produces a list of errors. If it succeeds, it produces a list of `Markdown.Block.Block` values, which can then go through a renderer, and be turned into `Element msg` values.

The next step is to define `elmUiRenderer` whose type is `Markdown.Renderer.Renderer (Element msg)`. This is actually a giant record of functions which produce an `Element msg` for every possible Markdown block. For example, it could look like this:

```
elmUiRenderer : Markdown.Renderer.Renderer (Element msg)
elmUiRenderer =
  { heading = heading
  , paragraph = paragraph [ spacing 15 ]
  , thematicBreak = none
  , text = text
  , strong = \content -> row [ Font.bold ] content
  , emphasis = \content -> row [ Font.italic ] content
  , codeSpan = code
  , link =
      \{ title, destination } body ->
        newTabLink
          [ htmlAttribute <| Html.Attributes.style "display" "inline-flex" ]
          { url = destination
          , label =
              paragraph [ Font.color color.blue ] body
          }
  , hardLineBreak = html <| Html.br [] []
  , image =
      \img ->
        image [ width fill ] { src = img.src, description = img.alt }
  , blockQuote =
      \children ->
        column
          [ padding 10
          , Border.widthEach { top = 0, right = 0, bottom = 0, left = 10 }
          , Border.color color.darkCharcoal
          , Background.color color.lightGrey
          ]
          children
  , unorderedList =
      \items ->
        column [ spacing 15 ]
          (items
            |> List.map
              (\(ListItem task children) ->
                row [ spacing 5 ]
                  [ row
                      [ alignTop ]
                      ((case task of
                        IncompleteTask ->
                          Input.defaultCheckbox False
```

```

        CompletedTask ->
            Input.defaultCheckbox True

        NoTask ->
            text "•"
    )
    :: text " "
    :: children
)
]
)
)
, orderedList =
    \startingIndex items ->
        column [ spacing 15 ]
        <| List.indexedMap
            (\index itemBlocks ->
                row [ spacing 5 ]
                [ row [ alignTop ]
                  <| text
                      (String.fromInt (index + startingIndex) ++ " ")
                      :: itemBlocks
                ]
            )
            items
, codeBlock = codeBlock
, html = Markdown.Html.oneOf []
, table = column []
, tableHeader = column []
, tableBody = column []
, tableRow = row []
, tableHeaderCell = \maybeAlignment children -> paragraph [] children
, tableCell = \maybeAlignment children -> paragraph [] children
}

```

The heading function could in turn be defined in this fashion:

```

heading :
  { level : Block.HeadingLevel, rawText : String, children : List (Element msg) }
  -> Element msg
heading { level, rawText, children } =
  paragraph
    [ Font.size
      (case level of
        Block.H1 ->
          36

        Block.H2 ->
          24

        _ ->
          20
      )
    , Font.bold
    , Region.heading <| Block.headingLevelToInt level
    ]
  children

```

As you can see, there is some wiring you have to do, but in return you get quite a lot of flexibility with regards to how you turn Markdown into an elm-ui layout.

Finally, we need to use the `markdownView` function in a layout, and render the result or errors accordingly:

```

main : Html msg
main =
  layout [ width fill ] <|
    case markdownView markdownBody of
      Ok renderedEls ->
        column [ spacing 30, padding 10, width fill ] renderedEls

      Err errors ->
        text errors

```

Additionally, this package allows you to handle custom HTML tags within Markdown strings by providing handlers in the `html` field of the renderer. In the example `elmUiRenderer` record above, this field was set to `Markdown.Html.oneOf []` so the renderer cannot handle



any HTML tags. It is beyond the scope of this guide but it's something you could find very useful.

## 6.4 Elm-markup

A possible alternative to `dillonkearns/elm-markdown` is `mdgriffith/elm-markup` which allows you to work with another text markup format. This package allows you to take marked up text, and transform it into output values of your choosing – for example, `elm-ui` elements.

Here is a sample of the markup:

```
|> Metadata
  title = Elmstatic now supports elm-markup
  tags = software other
```

```
*Elmstatic* now supports `elm-markup`{name}!
```

```
|> H2
  Let's look at some /code/
```

Code is highlighted using `[Highlight.js]{link | url = https://highlightjs.org}`:

```
|> Code
  lang = elm
  code =
    view =
      \content ->
        { title = ""
          , body = [ htmlTemplate content.siteTitle <| view content ]
        }
```

In addition to the package, there is a VS Code extension which provides syntax highlighting and checking for `.emu` files containing this markup.

When a document is parsed, you get parsing errors describing what's wrong. The markup is highly customisable: in your Elm code, you define a parser for all of the blocks starting

with an arrow (`|> Metadata`, `|> H2` and so on), as well as the inline formatting (such as `[Highlight.js]{link | url = https://highlightjs.org}`).

So how do we turn an elm-markup document into `Element msg` values? In the markup example above, there are many different pieces:

- Named blocks with key-value attributes such as `|> Metadata` and `|> Code`, or without attributes, such as `|> H1`
- Built-in styling syntax such as `*Elmstatic*` and `/code/`
- An annotated verbatim sequence ``elm-markup`{name}`
- An annotated sequence `[Highlight.js]{link | url = https://highlightjs.org}` which is used to define a link.

One more structure that elm-markup is able to parse is a tree – useful for dealing with nested lists, for example – but we’re going to leave that out to keep things simple. You can see how lists are parsed in the example code for the package.

The parser for a document with all these pieces is composed out of different “blocks”. Let’s start with only handling text with built-in styling:

```
import Mark exposing (Block, Document)
import Mark.Error

styledText : { bold : Bool, italic : Bool, strike : Bool } -> String -> Element msg
styledText styles str =
  let
    when cond value list =
      if cond then
        value :: list

      else
        list
  in
    text str
    |> el
    ([
```

```

|> when styles.bold Font.bold
|> when styles.italic Font.italic
|> when styles.strike Font.strike
)

```

```

inlineMarkup : Block (List (Element msg))
inlineMarkup =
    Mark.text styledText

```

`Mark.text` takes a function that defines how a string with some styles attached gets converted into output:

```

text : (Styles -> String -> result) -> Block (List result)

```

`elm-markup` parser functions return `Block` values which can be composed in order to handle more complex documents. The `text` function produces a block containing a list of output values, rather than a single value. Since `styledText` converts a style record and a string into a `el` with some font attributes, the type of output values is `Element msg`.

Then, we need to create a document. A document generally combines multiple parsing elements, but we are going to start out with only one element (`inlineText`) for now:

```

document : Document (Element msg)
document =
    Mark.document (paragraph []) inlineMarkup

```

With this, we can parse simple documents like this:

Let's look at some *\*styled\** /markup/.

Let's add a main function which parses the document and deals with possible errors:

```

markup : String
markup =
    """
Let's look at some *styled* /markup/.
    """

main : Html msg
main =
    let
        errorsToEl errors =
            errors
            |> List.map (Mark.Error.toString >> text)
            |> List.map (\txtEl -> paragraph [] [ txtEl ])
            |> textColumn []

        markupToEl =
            case Mark.compile document markup of
                Mark.Success el ->
                    el

                Mark.Almost { result, errors } ->
                    -- This is the case where there has been an error,
                    -- but it has been caught by `Mark.onError` and is still rendereable.
                    column []
                        [ errorsToEl errors
                        , result
                        ]

                Mark.Failure errors ->
                    errorsToEl errors

    in
        layout [ padding 10 ] markupToEl

```

In the case expression inside `markupToEl`, you can see that there are *three* possible outcomes: - If parsing succeeds, we get an `Element msg` value - If parsing fails, we get a list of errors which can be converted to strings with `Mark.Error.toString` - Finally, parsing can succeed *partially*, returning both a set of errors and a parsing result, in which case we display both.

If parsing succeeds, we should see a rendered result:

Let's look at some **styled markup**.

Moving on to other markup, let's deal with `|> H1`. The simplest parser definition for it would look like this:

```
Mark.block "H1" text Mark.string
```

This says that the contents of an H1 block should be parsed with the primitive parser `Mark.string` (producing a string), and then turned into an `Element msg` by wrapping with `Element.text`.

However, we would like to allow our headings to contain styling, so we need something more sophisticated than `Mark.string`. Luckily, we already have a parser for styled text (`inlineMarkup`) and `elm-markup` allows us to compose parsers so we can reuse it:

```
heading1 : Block (Element msg)
heading1 =
    Mark.block "H1"
        (\children ->
            paragraph [ Font.size 28, Region.heading 1 ] children
        )
    inlineMarkup
```

The `block` function takes a string (the name of the block), a function which converts the content into an output value, and a `Block content` value which describes how to parse the content:

```
block : String -> (content -> result) -> Block content -> Block result
```

In our case, the content type is `List (Element msg)` because the type of `inlineMarkup` is `Block (List (Element msg))`. Correspondingly, the anonymous function which is the

second argument takes a `List (Element msg)` and wraps it in `paragraph` to produce an output value.

We also need to declare that the markup can contain a mix of headings and styled text. For that, we can use `Mark.manyOf`, which converts a list of blocks into a single block:

```
document : Document (Element msg)
document =
  Mark.document (column []) <|
    Mark.manyOf
      [ inlineMarkup
        , heading1
      ]
```

The above definition is not quite right though. `inlineMarkup` is a `Block (List (Element msg))` while `heading1` is a `Block (Element msg)`. We can fix the discrepancy with `Mark.map`:

```
document : Document (Element msg)
document =
  Mark.document (column [ spacing 20 ]) <|
    Mark.manyOf
      [ Mark.map (paragraph []) inlineMarkup
        , heading1
      ]
```

Now, we can parse a document with a combination of styled text fragments and headings:

Let's look at some *\*styled\** /markup/.

```
|> H1
  This is a /heading/ with some inline styling
```

Let's look at some *styled markup*.

This is a *heading* with some inline styling

Now how about the `|> Metadata` block at the start of a document? Instead of text, it contains a record of keys and values:

```
|> Metadata
  title = Elmstatic now supports elm-markup
  tags = software other
```

It can be handled like this:

```
metadata : Block (Element msg)
metadata =
  Mark.record "Metadata"
    frontMatter
    |> Mark.field "title" Mark.string
    |> Mark.field "tags" Mark.string
    |> Mark.toBlock

frontMatter : String -> String -> Element msg
frontMatter title tagString =
  let
    tagsToEls tagStr =
      List.map
        (\tag ->
          el
            [ padding 5
              , Border.rounded 6
              , Background.color color.lightGrey
            ]
          <|
            text tag
        )
      <|
        String.split " " tagStr
  in
  column
    [ width fill
      , spacing 10
      , Border.widthEach { bottom = 1, top = 0, left = 0, right = 0 }
      , Border.color color.lightGrey
    ]
    [ paragraph [ Font.size 32, Region.heading 1 ] [ text title ]
```

```

    , row
      [ spacing 10
      , paddingEach { bottom = 10, top = 0, left = 0, right = 0 }
      ]
    <|
      tagsToEls tagString
  ]

```

In the `metadata` function, we use `Mark.record` to declare that `|> Metadata` is a specific kind of block containing key-value pairs. Then we describe the types of each field in the record, and finally finish things off by converting it to a block. The second argument of `metadata` is a function that receives the values of all the fields as its arguments, and uses them to produce an output value. In this case, `frontMatter` receives two strings (title and a string of tags) and converts them into an `Element msg`.

This works, but if we now throw the `metadata` block into the definition of `document`, there is a problem:

```

document : Document (Element msg)
document =
  Mark.document (column [ spacing 20 ]) <|
    Mark.manyOf
      [ metadata
      , Mark.map (paragraph []) inlineMarkup
      , heading1
      ]

```

This says that a `metadata` block can appear anywhere in the text of the document. However, we only want it to appear at the start. To enforce this rule, we need to switch from `Mark.document` to `Mark.documentWith` which caters for this scenario by handling `metadata` explicitly:



```

metadata : Block { title : String, tags : String }
metadata =
    Mark.record "Metadata"
        (\title tags -> { title = title, tags = tags })
        |> Mark.field "title" Mark.string
        |> Mark.field "tags" Mark.string
        |> Mark.toBlock

document : Document (Element msg)
document =
    Mark.documentWith
        (\meta body ->
            column [ width fill, spacing 20 ] <| frontMatter meta.title meta.tags :: body
        )
        { metadata = metadata
        , body =
            Mark.manyOf
                [ heading1
                , Mark.map (paragraph []) inlineMarkup
                ]
        }

```

There are a few changes to note here: \* metadata now describes a block that produces a record instead of an `Element msg` value. \* The first argument to `documentWith` is a function that receives the metadata (the record with `title` and `tags` in our case) and the body, which ends up being a `List (Element msg)` since all our parsing bits generate `Element msg` values and we combined them with `Mark.manyOf`. \* We now use `frontMatter` in this function to display the title and tags, instead of using it directly in metadata which we did when metadata was a regular block.

This document definition ensures that the markup starts with a metadata block, and if it doesn't, we will get a parsing error. The metadata is also used to render the title and tags:

# This is an elm-markup document

markup parsing

Let's look at some styled *markup*.

This is a *heading* with some inline styling

A code block is similar to a metadata block:

|> Code

```
lang = elm
code =
  view content =
    { title = ""
    , body =
      [ htmlTemplate content.siteTitle <| toHtml content ]
    }
```

```
code : Block (Element msg)
```

```
code =
```

```
  Mark.record "Code"
    (\lang str ->
      el
        [ padding 5
        , Background.color color.lightGrey
        , Border.rounded 6
        , fixedWidthFont
        ]
      <|
        text str
    )
  |> Mark.field "lang" Mark.string
  |> Mark.field "code" Mark.string
  |> Mark.toBlock
```

```
fixedWidthFont : Attribute msg
```

```
fixedWidthFont =
```

```
  Font.family [ Font.typeface "Courier New", Font.monospace ]
```

This takes care of a big chunk of the markup, but we still need to handle inline annotations like ``elm-markup`{name}` and `[Highlight.js]{link | url = https://highlightjs.org}`. For that, we need to revisit `inlineText`. Instead of `Mark.text`, we're going to use the more advanced `Mark.textWith`:

```
annotatedInlineMarkup =
  Mark.textWith
    { view = styledText
    , replacements = Mark.commonReplacements
    , inlines =
      [ Mark.annotation "link"
        (\texts url ->
          link [ Font.color color.blue ]
            { url = url
              , label =
                paragraph [] <|
                  List.map
                    (\( styles, str ) -> styledText styles str)
                    texts
              }
            )
          |> Mark.field "url" Mark.string
        , Mark.verbatim "name" (\str -> el [ fixedWidthFont ] <| text str)
      ]
    }
```

Instead of a single function taken by `Mark.text`, it takes a record. The first field, `view`, is a function that handles inline styles, so we reuse the already defined `styledText` here. `replacements` defines a map of character replacements (such as replacing two dashes with an em dash, or straight quotes with curly quotes). We use `Mark.commonReplacements` which is the default mapping supplied by the `elm-markup` package.

Finally, `inlines` is where we are going to deal with inline annotations. The structure of an annotation definition like `link` above is very similar to how we handled the record in the `|>` Metadata block. For verbatim annotations, we use `Mark.verbatim` to wrap the text in `el [ fixedWidthFont ]`.

code and `annotatedInlineMarkup` need to be added to the definition of `document`:

```

document : Document (Element msg)
document =
    Mark.documentWith
        (\meta body ->
            column [ spacing 20 ] <| frontMatter meta.title meta.tags :: body
        )
    { metadata = metadata
    , body =
        Mark.manyOf
            [ heading1
            , code
            , Mark.map (paragraph []) annotatedInlineMarkup
            ]
    }

```

As a result, we can display fairly sophisticated documents like the one below:

```

|> Metadata
    title = This is an elm-markup document
    tags = markup parsing

```

Let's look at some *\*styled\** /markup/.

```

|> H1
    This is a /header/ with some inline styling

```

This is a ``verbatim annotation`{name}``.

This is a `[link with *style*]{link | url = "#"}`.

```

|> Code
    lang = elm
    code =
        view content =
            { title = ""
            , body = [ htmlTemplate content.siteTitle <| toHtml content ]
            }

```

## This is an elm-markup document

markup parsing

Let's look at some styled *markup*.

This is a *heading* with some inline styling

This is a verbatim annotation.

This is a [link with style](#).

```
view content =
  { title = ""
  , body =
      [ htmlTemplate content.siteTitle <| toHtml content ]
  }
```

## 6.5 Iframes

Occasionally you might need to show external content in your application which has to appear in an `iframe`. `elm-ui` doesn't make available an element for `iframes` directly, however this is easily resolved via the `html` escape hatch:

```
main : Html msg
main =
  layout
    [ width fill, height fill, padding 50 ] <|
    el
      [ width fill
      , centerX
      , centerY
      , padding 20
      , Border.width 2
      , Border.color color.lightGrey
      ]
    <|
    html <|
      Html.iframe
```

```
[ Html.Attributes.src <| "pattern.html#Tabs"  
  , Html.Attributes.style "border" "none"  
  , Html.Attributes.style "width" "100%"  
  , Html.Attributes.style "height" "100%"  
]  
[]
```

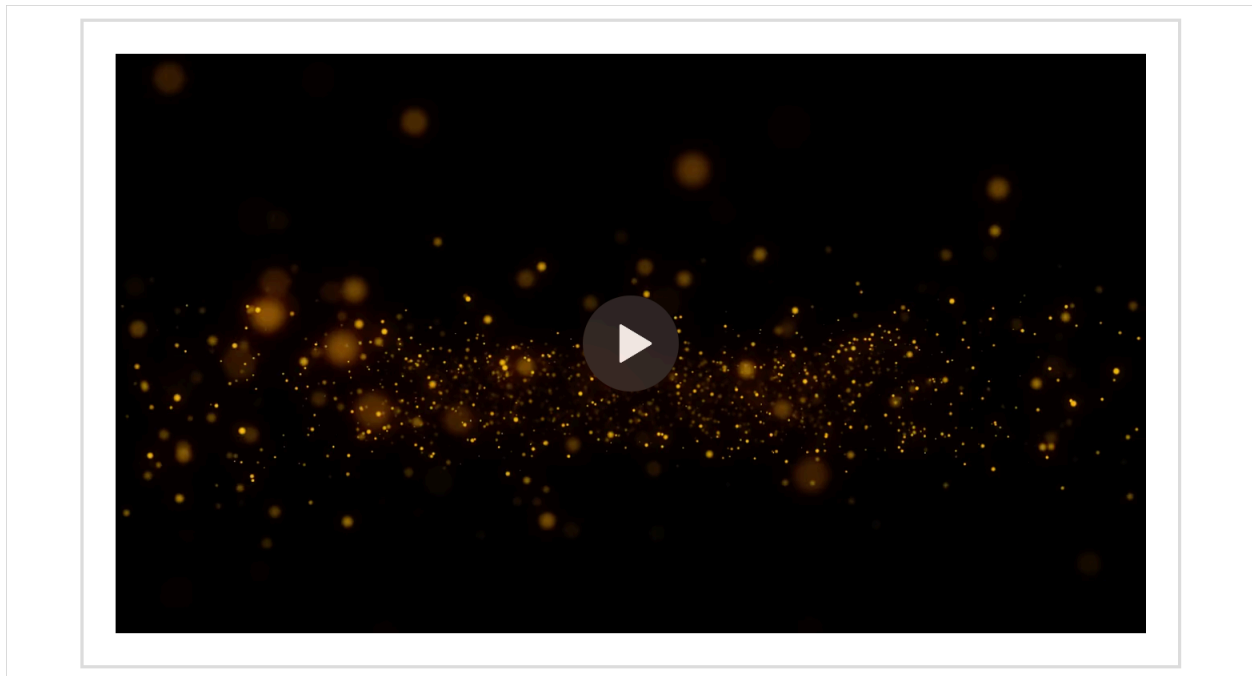


The `iframe` needs an `src` attribute, and you'll likely want to disable the default pseudo-3D border as well. Setting width and height to 100% allows you to size the `iframe` via the wrapper `Element` rather than with CSS.

## 6.6 Video

The situation with video is similar to `iframes`. You need to resort to using `html` and `Html.video`:

```
el
  [ width fill
  , padding 20
  , Border.width 2
  , Border.color color.lightGrey
  ]
<|
html <|
  Html.video
    [ Attr.src "video.mp4"
    , Attr.controls True
    , Attr.loop True
    , Attr.attribute "muted" ""
    , Attr.style "width" "100%"
    ]
  []
```



Sometimes, video is used as a dynamic background, and this is easily achievable with `behindContent`:

```
el
  [ width fill
    , height <| px 390
    , Border.width 2
    , Border.color color.lightGrey
    , behindContent <|
      html <|
        Html.video
          [ Attr.src "video.mp4"
            , Attr.controls False
            , Attr.loop True
            , Attr.autoplay True
            , Attr.preload "auto"
            , Attr.attribute "muted" ""
            , Attr.attribute "playsinline" ""
            , Attr.style "width" "100%"
          ]
        []
      ]
    ]
  <|
  el [ centerX, centerY, Font.color color.white, Font.size 64, Font.bold ] <|
    text "Background video"
```





## 6.7 Custom elements

Continuing with the theme of applying the `html` escape hatch, custom elements can be inserted into an `elm-ui` layout via `Html.node`. For example, let's suppose we have wrapped the popular Ace JS editor in an `<ace-editor>` custom element. We can then add an editor to our UI:

```
el [ width fill, height fill ] <|
  html <|
    Html.node "ace-editor"
      [ Html.Attributes.attribute "mode" "ace/mode/elm"
      , Html.Attributes.attribute "wrapmode" "true"
      , Html.Attributes.style "height" "500px"
      ]
    [ Html.text codeSample ]
```

```
1 module CustomElement exposing (..)
2
3 import Browser
4 import Element exposing (..)
5 import Element.Background as Background
6 import Element.Border as Border
7 import Element.Events exposing (..)
8 import Element.Font as Font
9 import Element.Input as Input
10 import Html exposing (Html)
11 import Html.Attributes
12
13
14 type alias Model =
15   String
16
17
18 type Msg
19   = UserTypedText String
20
21
22 view : Model -> Html Msg
23 view model =
24   layout [ padding 50 ] <|
25     el [ width fill, height fill ] <|
```

The custom element may generate events which we would like to turn into messages. If our element generates change events, they can be handled like this:

```

view : Model -> Html Msg
view model =
    let
        onChange =
            Json.Decode.string
                |> Json.Decode.at [ "target", "editorValue" ]
                |> Json.Decode.map UserTypedText
                |> Html.Events.on "change"
    in
        layout [ padding 20 ] <|
            column [ width fill, spacing 20 ]
                [ el [ width fill, height fill, htmlAttribute onChange ] <|
                    html <|
                        Html.node "ace-editor"
                            [ Html.Attributes.attribute "mode" "ace/mode/elm"
                              , Html.Attributes.attribute "wrapmode" "true"
                              , Html.Attributes.style "height" "500px"
                            ]
                        [ Html.text codeSample ]
                ]

```

We need to use the `Html.Events` module to define an `onChange` event handler, and then wrap it in `htmlAttribute` when adding it to the `el`. This will then produce `UserTypedText` messages with some data extracted from the events.

## 6.8 Drag and drop

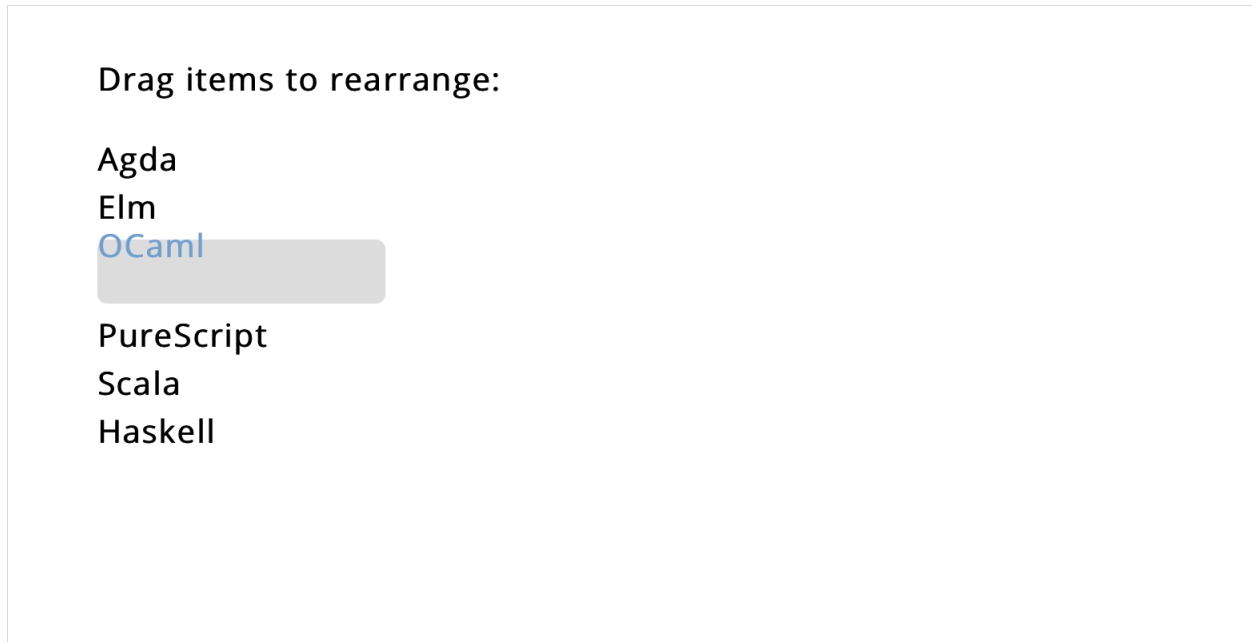
For drag-and-drop functionality, it's possible to combine the `annaghi/dnd-list` package with `elm-ui`. Even though it says “list” in the package name, checking out the [examples<sup>2</sup>](https://annaghi.github.io/dnd-list/gallery) page for the package demonstrates that it can be used for a variety of user interfaces. However, in this guide we will indeed look at implementing a list which can be reordered via drag-and-drop.

As you will see, there is a bit of friction in that this package requires adding CSS styles and HTML IDs to elements, which isn't something you'd do in “pure” `elm-ui`, but nevertheless it's a very practical solution.

---

<sup>2</sup><https://annaghi.github.io/dnd-list/gallery>

So let's create a list where items can be reordered via dragging. In the image below, programming languages can be reordered, and the OCaml item is in the process of being dragged:



We'll start by defining some types:

```
import DnDList

type alias Lang =
    String

type alias Model =
    { dnd : DnDList.Model
    , items : List Lang
    }

type Msg
    = ListMsg DnDList.Msg
```

We need to store some drag-and-drop state in the model, and we need to handle drag-and-drop messages (wrapped in ListMsg).

The drag-and-drop functionality has to be configured by supplying a config record and a message constructor to the `DnDList.create` function:

```
config : DnDList.Config Lang
config =
  { beforeUpdate = \_ _ list -> list
  , movement = DnDList.Free
  , listen = DnDList.OnDrag
  , operation = DnDList.Rotate
  }
```

```
dndList : DnDList.System Lang Msg
dndList =
  DnDList.create config ListMsg
```

The value produced by `create` is a record which is then used to initialise the model and set up subscriptions, among other things:

```
init : () -> ( Model, Cmd Msg )
init _ =
  ( { dnd = dndList.model
    , items = [ "Agda", "Elm", "Haskell", "OCaml", "PureScript", "Scala" ]
    }
  , Cmd.none
  )
```

```
subscriptions : Model -> Sub Msg
subscriptions model =
  dndList.subscriptions model.dnd
```

Next, we can construct the view:

```

view : Model -> Html.Html Msg
view model =
  layout
    [ width fill
    , height fill
    , padding 50
    , inFront <| ghostView model.dnd model.items
    ]
  <|
    column [ width <| px 200, padding 10, spacing 10, Font.bold ] <|
      ((el [ height <| px 40 ] <| text "Drag items to rearrange:")
       :: List.indexedMap (itemView model.dnd) model.items
      )

```

There are two principal parts of the view. One is the column of items, each rendered with `itemView`. The other is a `ghostView` which is rendered on top of the list. The ghost view displays the item that's being dragged:

```

ghostView : DnDList.Model -> List Lang -> Element Msg
ghostView dndModel items =
  let
    maybeDragItem : Maybe Lang
    maybeDragItem =
      dndList.info dndModel
      |> Maybe.andThen
        (\{ dragIndex } -> items |> List.drop dragIndex |> List.head)
  in
  case maybeDragItem of
    Just item ->
      el
        ([ Font.bold, Font.color color.blue ]
         ++ htmlAttrs (dndList.ghostStyles dndModel)
        )
      <|
        text item

    Nothing ->
      none

```

We use the function in `dndList.info` to get a `Maybe` value with the details of the dragged

item. If there is no drag operation in progress, the function returns `Nothing`. Otherwise, we get a value to display:

```
el
  ([ Font.bold, Font.color color.blue ]
    ++ htmlAttrs (dndList.ghostStyles dndModel)
  )
<|
  text item
```

Importantly, we need to add a set of CSS attributes returned by `dndList.ghostStyles` to this element. `htmlAttrs` is a function that wraps a list of `Html.Attribute Msg` values in `htmlAttribute` to convert them to elm-ui's `Attribute Msg` values:

```
htmlAttrs : List (Html.Attribute Msg) -> List (Attribute Msg)
htmlAttrs =
  List.map htmlAttribute
```

The styles returned by `ghostStyles` position the element relative to the viewport:

```
{
  position: fixed;
  left: 0;
  top: 0;
  transform: translate3d(the vector is calculated from the dragElement
    and the mouse position in pixels);
  height: the dragElement's height in pixels;
  width: the dragElement's width in pixels;
  pointer-events: none;
}
```

Onwards to `itemView`, where several cases need to be handled:

- If there is no drag operation in progress, then the item needs to have drag events attached
- If there is a drag operation and the item is the one that's being dragged, then a placeholder needs to be shown in its drop position

- Finally, all other items need to have drop events attached when a drag operation is ongoing.

The implementation of `itemView` looks like this:

```

itemView : DnDList.Model -> Int -> Lang -> Element Msg
itemView dndModel index item =
    let
        itemId : String
        itemId =
            "id-" ++ item
    in
    case dndList.info dndModel of
    Just { dragIndex } ->
        if dragIndex /= index then
            el
                ([ width fill ]
                 ++ htmlAttrs
                 (Html.Attributes.id itemId
                  :: dndList.dropEvents index itemId
                 )
                )
            <|
                text item
        else
            el
                [ width fill
                , height fill
                , padding 10
                , Background.color color.lightGrey
                , Border.rounded 6
                , htmlAttribute (Html.Attributes.id itemId)
                ]
            <|
                text " "
    Nothing ->
        el
            (htmlAttrs <|
             Html.Attributes.id itemId
             :: dndList.dragEvents index itemId
            )

```

```

    )
  <|
  text item

```

Note that the package also requires list items to have unique ID attributes which in this case are simply generated from item text.

Finally, there is a bit of wiring needed in the update function to handle drag-and-drop messages and commands:

```

update : Msg -> Model -> ( Model, Cmd Msg )
update message model =
  case message of
    ListMsg msg ->
      let
        ( dnd, items ) =
          dndList.update msg model.dnd model.items
      in
        ( { model | dnd = dnd, items = items }
        , dndList.commands dnd
        )

```



## 7. Organising UI code

As `elm-ui` code is ultimately just regular Elm code, you can organise it just like you would any other code, so in this section I would only like to suggest a few options for structuring the UI code without being prescriptive.

There are two primary questions that arise:

- how to reuse portions of the UI code that are shared between parts of the view?
- how to deal with the parts of the UI which depend on the state in the model?

Let's consider both of them.

### 7.1 Structuring the code for reuse

The reusable parts of UI code will consist primarily of groups of attributes and groups of elements. For example, you might want to have a set of attributes applied to every button:

```
[ padding 10
, Border.width 3
, Border.rounded 6
, Border.color blue
, Background.color lightBlue
]
```

You might also have a reusable page element made up of several `elm-ui` elements and parameterised in some way:

```

logo : Color -> Int -> List (Attribute msg) -> Element msg
logo color fontSize attrs =
    row (Font.size fontSize :: attrs)
        [ image [ width <| px 100 ] { src = "/img/logo.png", description = "Logo" }
          , el [ width <| px 10 ] <| text " "
          , el [ Font.color color ] <| text "COMPANY"
        ]

```

You'll also have things like colours and font attributes that are used all over the code base. The simplest approach to manage these is to give names to all the reusable values and functions:

```

import Element exposing (..)
import Element.Background as Background
import Element.Border as Border
import Element.Font as Font
import Element.Input as Input

```

```

blue =
    rgb255 0x72 0x9F 0xCF

```

```

lightBlue =
    rgb255 0xC5 0xE8 0xF7

```

```

lightGrey =
    rgb255 0xE0 0xE0 0xE0

```

```

white =
    rgb255 0xFF 0xFF 0xFF

```

```

buttonAttrs =
    [ padding 10
    , Border.width 3
    , Border.rounded 6
    , Border.color blue
    , Background.color lightBlue
    , mouseDown
      [ Background.color blue

```

```

        , Border.color blue
        , Font.color white
    ]
    , mouseOver
    [ Background.color white
      , Border.color lightGrey
    ]
]

linkAttrs =
    [ Font.bold, Font.underline, Font.color blue ]

logo : Color -> Int -> List (Attribute msg) -> Element msg
logo color fontSize attrs =
    row (Font.size fontSize :: attrs)
        [ image [ width <| px 100 ] { src = "/img/logo.png", description = "Logo" }
          , el [ width <| px 10 ] <| text " "
          , el [ Font.color color ] <| text "COMPANY"
        ]

main =
    layout [] <|
        column []
            [ -- A button with an additional attributes
              Input.button (buttonAttrs ++ [ Font.variant Font.smallCaps ])
                { onPress = Just True, label = text "Button 1" }

              -- A button with an attribute that overrides a predefined one
              , Input.button (buttonAttrs ++ [ Background.color lightGrey ])
                { onPress = Just True, label = text "Button 2" }

              -- A link with attributes
              , link linkAttrs { url = "#", label = text "Styled link" }

              -- A link which looks like a button due to having button attributes
              , link buttonAttrs { url = "#", label = text "Button-like link" }

              -- Logo instance
              , logo white 20 []
            ]

```

You will likely also want to define the reusable bits in a separate module, for example `Ui.elm`.

Another option is to use records to organise the UI code:

```
import Element exposing (..)
import Element.Background as Background
import Element.Border as Border
import Element.Font as Font
import Element.Input as Input

color =
  { blue = rgb255 0x72 0x9F 0xCF
  , lightBlue = rgb255 0xC5 0xE8 0xF7
  , lightGrey = rgb255 0xE0 0xE0 0xE0
  , white = rgb255 0xFF 0xFF 0xFF
  }

attrs =
  { button =
    [ padding 10
    , Border.width 3
    , Border.rounded 6
    , Border.color color.blue
    , Background.color color.lightBlue
    , mouseDown
      [ Background.color color.blue
      , Border.color color.blue
      , Font.color color.white
      ]
    , mouseOver
      [ Background.color color.white
      , Border.color color.lightGrey
      ]
    ]
  , link = [ Font.bold, Font.underline, Font.color color.blue ]
  }

element =
  { logo =
    \givenColor givenFontSize givenAttrs ->
```

```

    row (Font.size givenFontSize :: givenAttrs)
      [ image [ width <| px 100 ] { src = "/img/logo.png", description = "Logo" }
      , el [ width <| px 10 ] <| text " "
      , el [ Font.color givenColor ] <| text "COMPANY"
      ]
  }

main =
  layout [] <|
    column []
      [ -- A button with an additional attributes
        Input.button (attrs.button ++ [ Font.variant Font.smallCaps ])
          { onPress = Just True, label = text "Button 1" }

        -- A button with an attribute that overrides a predefined one
        , Input.button (attrs.button ++ [ Background.color color.lightGrey ])
          { onPress = Just True, label = text "Button 2" }

        -- A link with attributes
        , link attrs.link { url = "#", label = text "Styled link" }

        -- A link which looks like a button due to having button attributes
        , link attrs.button { url = "#", label = text "Button-like link" }

        -- Logo instance
        , element.logo color.white 20 []
      ]

```

By combining modules and records, you can logically structure the reusable UI elements.

Using named attribute groups is conceptually similar to applying CSS classes to your elements. For an example of this approach, have a look at the [Orasund/elm-ui-framework package](https://package.elm-lang.org/packages/Orasund/elm-ui-framework/latest)<sup>1</sup>.

For more complex scenarios where you need to enforce UI conventions and avoid visual bugs, you could create a domain-specific API on top of `elm-ui`. For example, let's say you wanted to have different types of links: some of them would be regular hyperlinks, and others would look like buttons. Additionally, they would be styled differently depending on whether they

<sup>1</sup><https://package.elm-lang.org/packages/Orasund/elm-ui-framework/latest>

appear in the navigation or the content of a page. You could then create a `Link` module which implements these requirements:

```
module Link exposing (Link, Role(..), Theme(..), asElement, link, withRole, withTheme)

import Color
import Element exposing (..)
import Element.Background as Background
import Element.Border as Border
import Element.Font as Font
import Html.Attributes

type Link msg
  = Link { url : String, label : String, role : Role, theme : Theme }

type Role
  = Hyperlink
  | Button

type Theme
  = Body
  | Navigation

link : String -> String -> Link msg
link url label =
  Link { url = url, label = label, role = Hyperlink, theme = Body }

withRole : Role -> Link msg -> Link msg
withRole role (Link givenLink) =
  Link { givenLink | role = role }

withTheme : Theme -> Link msg -> Link msg
withTheme theme (Link givenLink) =
  Link { givenLink | theme = theme }

asElement : Link msg -> Element msg
```

```
asElement (Link givenLink) =
  let
    attrs =
      case ( givenLink.role, givenLink.theme ) of
        ( Hyperlink, Body ) ->
          [ Font.bold, Font.underline, Font.color Color.blue ]

        ( Hyperlink, Navigation ) ->
          [ Font.bold, Font.color Color.lightBlue ]

        ( Button, Body ) ->
          [ Border.width 3
            , Border.rounded 6
            , Border.color Color.blue
            , Background.color Color.lightBlue
          ]

        ( Button, Navigation ) ->
          [ Border.width 3
            , Border.rounded 6
            , Border.color Color.lightBlue
            , Background.color Color.lightGrey
          ]
    in
    Element.link attrs { url = givenLink.url, label = text givenLink.label }
```

In this module, I'm following an API convention suggested in [this talk by Brian Hicks<sup>2</sup>](#) but you could of course choose a different style. With such an approach, you could build your UI using domain-specific functions and types rather than plain `elm-ui`:

---

<sup>2</sup><https://www.youtube.com/watch?v=PDyWP-0H4Zo>

```

import Element
import Link exposing (..)

main =
  Element.layout [] <|
    Element.column []
      [ link "https://example.com" "Regular link"
        |> asElement
      , link "https://example.com" "Button link"
        |> withRole Link.Button
        |> asElement
      , link "https://example.com" "Navigation link"
        |> withTheme Link.Navigation
        |> asElement
      ]

```

## 7.2 Stateful UIs

The stereotypical example of parameterised UI is implementing light and dark themes for your application. Another example is having different layouts for mobile and desktop.

Of course, there's nothing stopping you from simply branching on some field in the model:

```

view model =
  let
    linkAttrs =
      case model.theme of
        Light ->
          [ Font.underline, Font.color blue ]

        Dark ->
          [ Font.bold, Font.underline, Font.color white ]

  in
    ...

```

By extracting functions, you can get pretty far with this approach. However, if you have a lot of functions the view is composed of, it can become cumbersome to pass the necessary



parameters into all of them. `miniBill/elm-ui-with-context` package offers a solution which mimics the interface of `elm-ui` but makes a “context” value given to the `layout` function available to all of them. This is best explained with an example.

Let’s see what support for themes looks like when using this package. First, we need to import the package modules instead of `elm-ui` modules:

```
import Element.WithContext exposing (..)
import Element.WithContext.Background as Background
import Element.WithContext.Border as Border
import Element.WithContext.Events as Events
import Element.WithContext.Font as Font
import Element.WithContext.Input as Input
```

Then, we can define the data that forms the “context” and add it to the model:

```
type Theme
  = Light
  | Dark

type alias Context =
  { theme : Theme }

type alias Model =
  { context : Context }
```

The package documentation recommends only placing constant or rarely changing parameters into the context. So a theme is fine, but the viewport size isn’t. This is because changes to the context will cause the recalculation of all the `lazy` nodes.

Next, let’s add a couple of helper functions which return a colour for a given theme:

```

fontColor : Theme -> Color
fontColor theme =
  case theme of
    Light ->
      color.darkCharcoal

    Dark ->
      color.white

backgroundColor : Theme -> Color
backgroundColor theme =
  case theme of
    Light ->
      color.blue

    Dark ->
      color.lightBlue

```

Finally, here is the interesting part:

```

view : Model -> Html Msg
view model =
  layout model.context [ padding 50 ] <|
    column [ spacing 20 ]
      [ button "No context button"
      ]

button : String -> Element Context Msg
button label =
  Input.button
    [ withAttribute (\context -> backgroundColor context.theme) Background.color
    , withAttribute (\context -> fontColor context.theme) Font.color
    , Border.width 3
    , Border.color color.blue
    ]
    { onPress = Just UserPressedButton, label = text label }

```

We need to pass the context value to layout, but note that we *do not* pass it to button, our helper function that wraps Input.button. Yet, in button we are able to use context to

set attribute values via the `withAttribute` function from the package. There is a similar function `with` for elements.

With a crafty definition of type aliases, it's also possible to switch to this package without needing to modify your UI code, but it does require using qualified functions names to begin with.

A potential downside of using this package is that it is an additional dependency that needs to be kept in sync with the API changes in `elm-ui`.