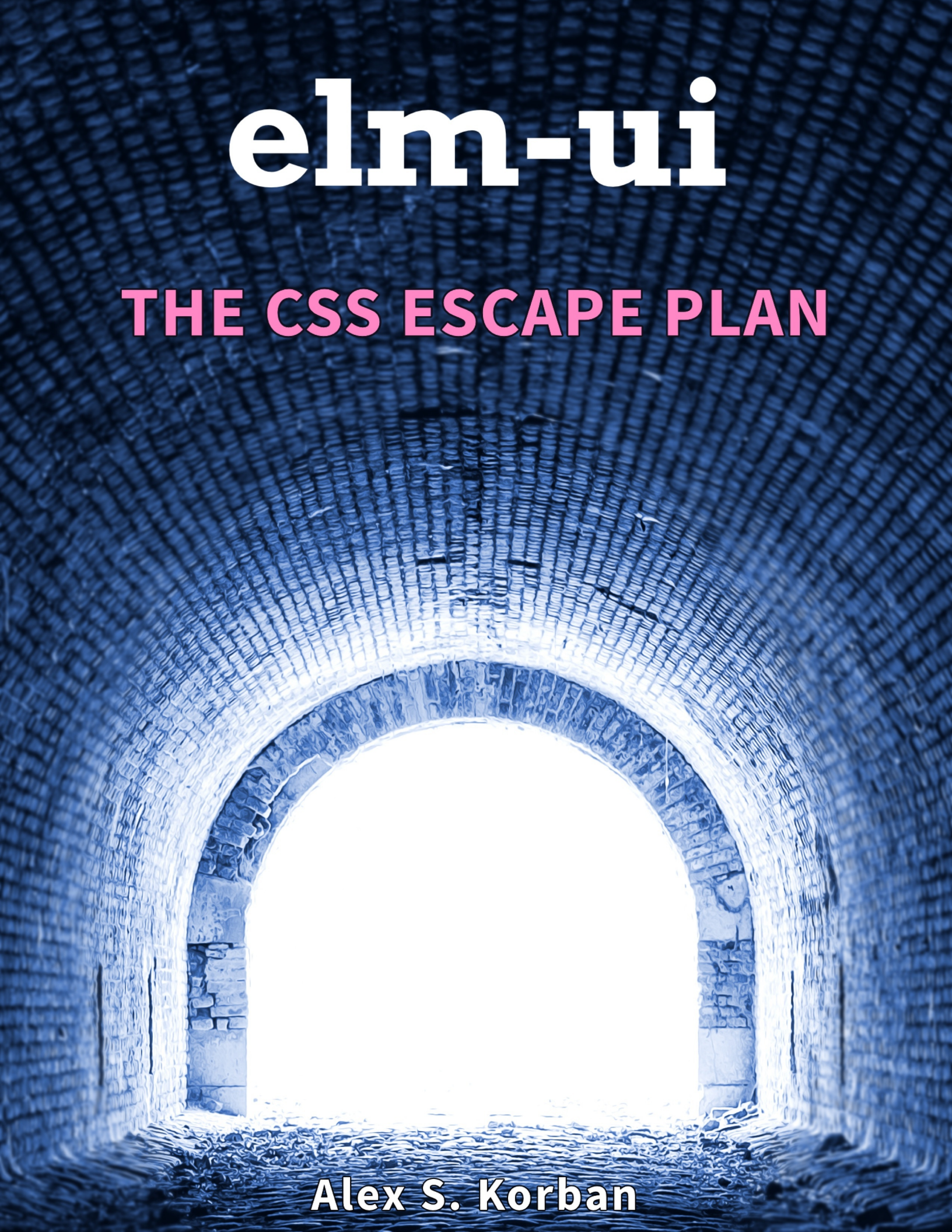

elm-ui
THE CSS ESCAPE PLAN
Alex S. Korban

elm-ui: The CSS Escape Plan

Alex S. Korban

Contents

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.

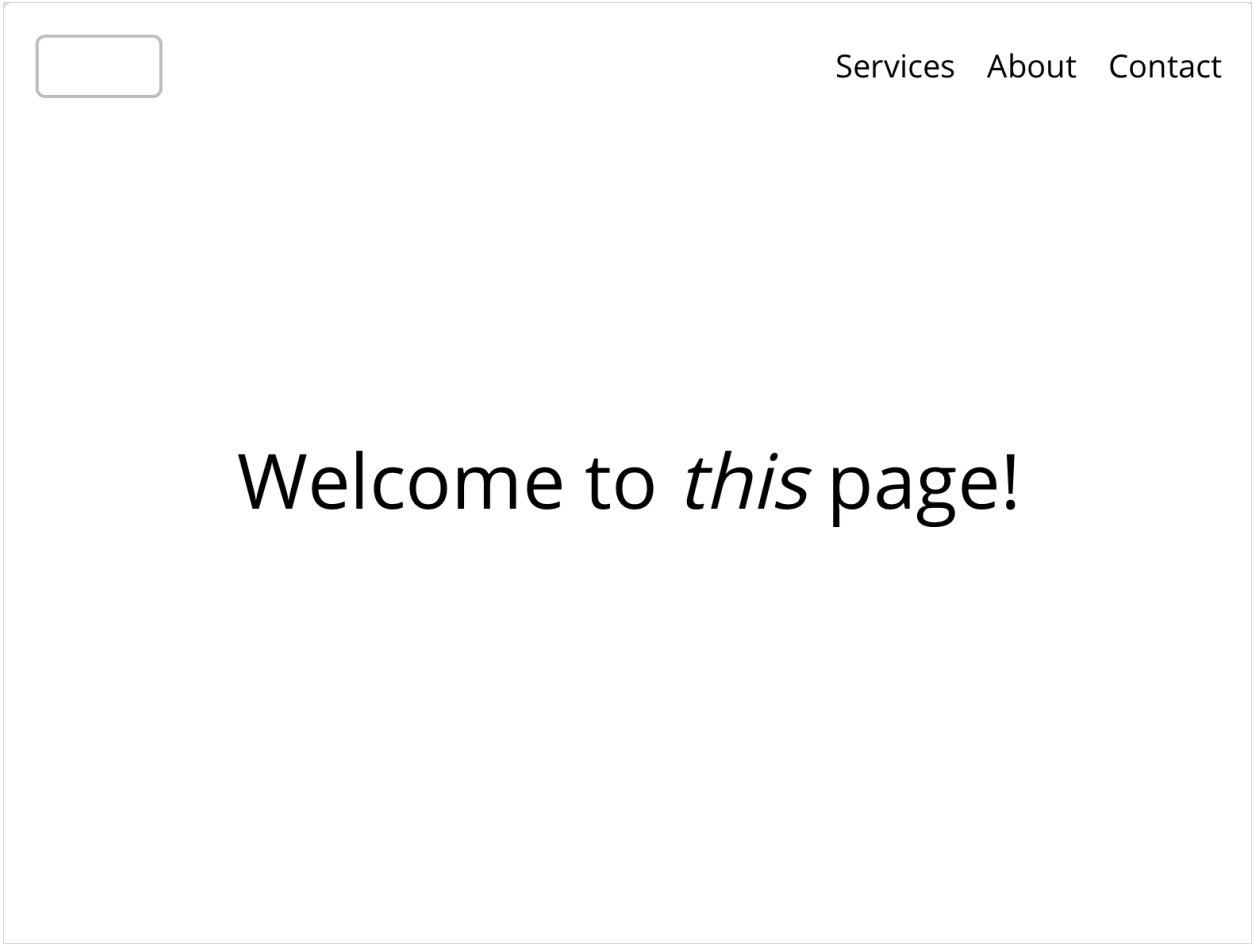
The guide is divided into the following sections:

- 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
                                            --   horizontally & vertically
                , padding 50                -- 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.

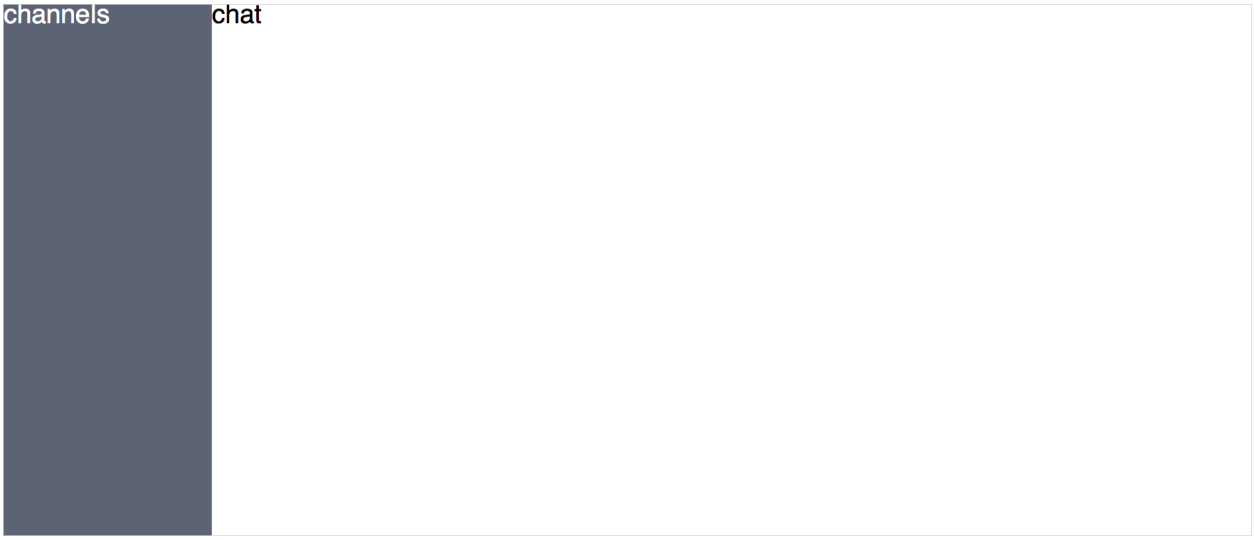
2.2 Going further: a chat page layout

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:

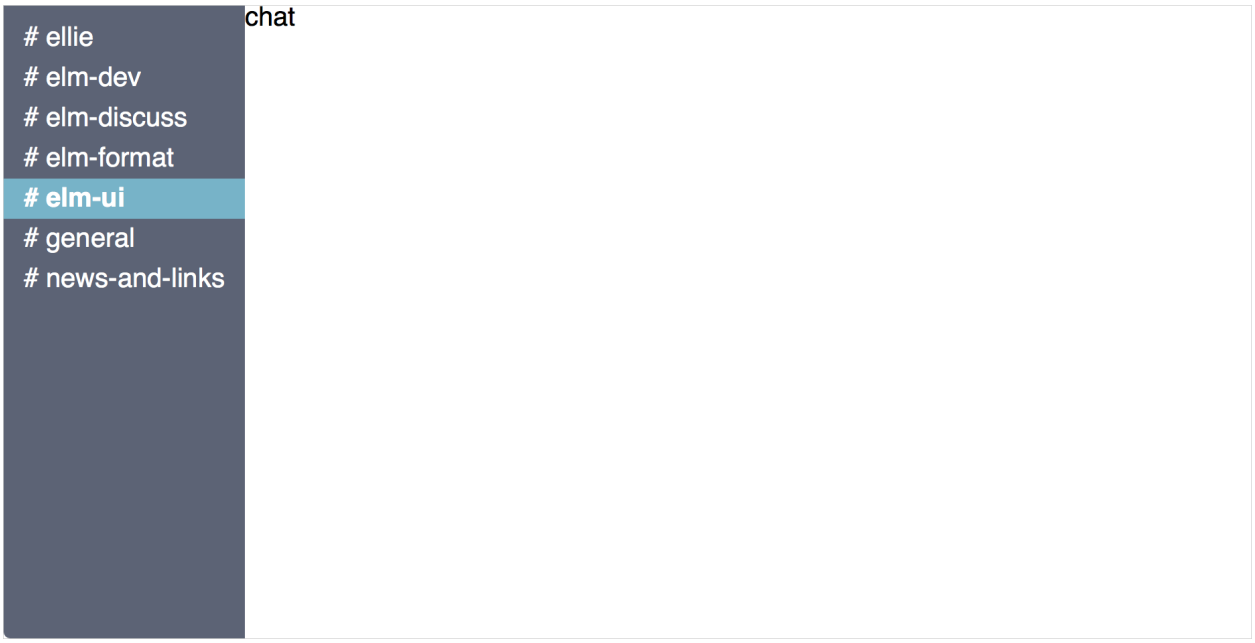
```
import Element.Background as Background
import Element.Border as Border
```

`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:

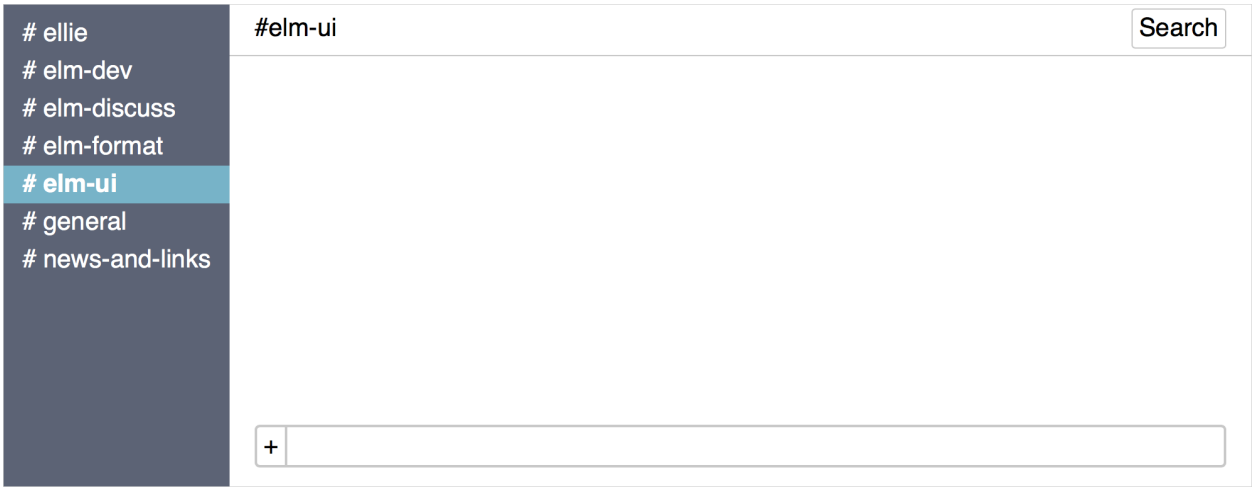
```
channelPanel : List String -> String -> Element msg
channelPanel channels activeChannel =
    let
        activeChannelAttrs =
            [ Background.color <| rgb255 117 179 201, Font.bold ]

        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
        ]
```

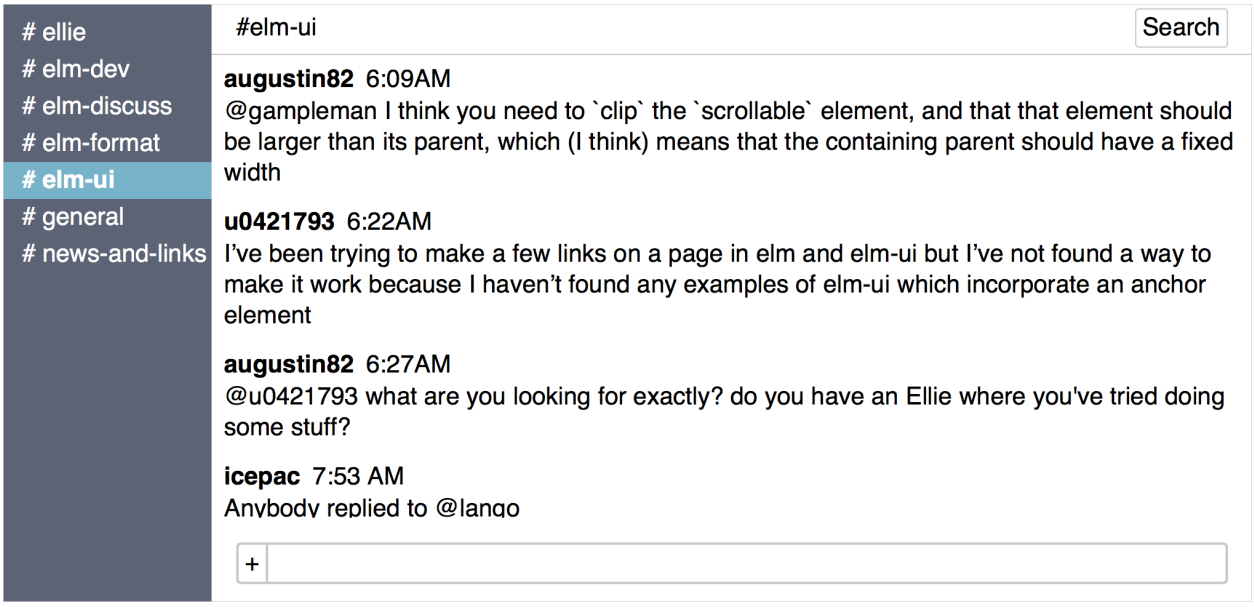
At first glance, this might look like a lot of code, however note that it's really straightforward, and most of it is simply layout and styling attributes. At the bottom of the function, it's very clearly stated that we have vertically arranged header, messages and footer.

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.