Если вы видите что-то необычное, просто сообщите мне.

Elm Part 4: Navigation

In part 3 of this series, we learned a few more complexities about how Elm works. We examined how to bridge Elm types and Haskell types using the elm-bridge library. We also saw a couple basic ways to incorporate effects into our Elm application. We saw how to use a random generator and how to send HTTP requests.

These forced us to stretch our idea of what our program was doing. Our original Todo application only controlled a static page with the sandbox function. But this new program used element to introduce effects into our program structure.

But there's still another level for us to get to. Pretty much any web app will need many pages, and we haven't seen what navigation looks like in Elm. To conclude this series, let's see how we incorporate different pages. We'll need to introduce a couple more components into our application for this.

To see the code for this part in action, check out our Github repository! You'll want to look at the ElmNavigation folder.

SIMPLE NAVIGATION

Now you might be thinking navigation should be simple. After all, we can use normal HTML elements on our page, including the a element for links. So we'd set up different HTML files in our project and use routes to dictate which page to visit. Before Elm 0.19, this was all you would do.

But this approach has some key performance weaknesses. Clicking a link will always lead to a page refresh which can be disrupting for the user. This approach will also lead us to do a lot of redundant loading of our library code. Each new page will have to reload the generated Javascript for Data.String, for example. The latest version of Elm has a new solution for this that fits within the Elm architecture.

AN APPLICATION

In our previous articles, we described our whole application using the element function. But now it's time to evolve from that definition. The application function provides us the tools we need to build something bigger. Let's start by looking at its type signature (see the appendix at the bottom for imports):

```
application :
  { init : flags -> Url -> Key -> (model, Cmd msg)
  , view : model -> Document msg
  , update : msg -> model -> (model, Cmd msg)
  , subscriptions : model -> Sub msg
  , onUrlRequest : UrlRequest -> msg
  , onUrlChange : Url -> msg
  }
  -> Program flags model msg
```

There are a couple new fields to this application function. But we can start by looking at the changes to what we already know. Our init function now takes a couple extra parameters, the Url and the Key. Getting a Url when our app launches means we can display different content depending on what page our users visit first. The Key is a special navigation tool we get when our app starts that helps us in routing. We need it for sending our own navigation commands.

Our view and update functions haven't really changed their types. All that's new is the view produces Document instead of only Html. A Document is a wrapper that lets us add a title to our web page, nothing scary. The subscriptions field has the same type (and we'll still ignore it for the most part).

This brings us to the new fields, onUrlRequest and onUrlChange. These intercept events that can change the page URL. We use onUrlChange to update our page when a user changes the URL at the top bar. Then we use onUrlRequest to deal with a links the user clicks on the page.

BASIC SETUP

Let's see how all these work by building a small dummy application. We'll have three pages, arbitrarily titled "Contents", "Intro", and "Conclusion". Our content will just be a few links allowing us to navigate back and forth. Let's start off with a few simple types. For our program state, we store the URL so we can configure the page we're on. We also store the navigation key because we need it to push changes to the page. Then for our messages, we'll have constructors for URL requests and changes:

```
type AppState = AppState
  { url: Url
  , navKey : Key
  }

type AppMessage =
  NoUpdate |
  ClickedLink UrlRequest |
  UrlChanged Url
```

When we initialize this application, we'll pass the URL and Key through to our state. We'll always start the user at the contents page. We cause a transition with the pushUrl command, which requires we use the navigation key.

```
appInit : () -> Url -> Key -> (AppState, Cmd AppMessage)
appInit _ url key =
  let st = AppState {url = url, navKey = key}
  in (st, pushUrl key "/contents")
```

UPDATING THE URL

Now we can start filling in our application. We've got message types corresponding to the URL requests and changes, so it's easy to fill those in.

```
main : Program () AppState AppMessage
main = Browser.application
  { init : appInit
  , view = appView
  , update = appUpdate
```

```
  , subscriptions = appSubscriptions
  , onUrlRequest = ClickedLink -- Use the message!
  , onUrlChanged = UrlChanged
  }
```

Our subscriptions, once again, will be Sub.none. So we're now down to filling in our update and view functions.

The first real business of our update function is to handle link clicks. For this, we have to break the UrlRequest down into its Internal and External cases:

```
appUpdate : AppMessage -> AppState -> (AppState, Cmd AppMessage)
appUpdate msg (AppState s) = case msg of
  NoUpdate -> (AppState s, Cmd.none)
  ClickedLink urlRequest -> case urlRequest of
    Internal url -> ...
    External href -> ...
```

Internal requests go to pages within our application. External requests go to other sites. We have to use different commands for each of these. As we saw in the initialization, we use pushUrl for internal requests. Then external requests will use the load function from our navigation library.

```
appUpdate : AppMessage -> AppState -> (AppState, Cmd AppMessage)
appUpdate msg (AppState s) = case msg of
  NoUpdate -> (AppState s, Cmd.none)
  ClickedLink urlRequest -> case urlRequest of
    Internal url -> (AppState s, pushUrl s.navKey (toString url))
    External href -> (AppState s, load href)
```

Once the URL has changed, we'll have another message. The only thing we need to do with this one is update our internal state of the URL.

```
appUpdate : AppMessage -> AppState -> (AppState, Cmd AppMessage)
appUpdate msg (AppState s) = case msg of
  NoUpdate -> (AppState s, Cmd.none)
  ClickedLink urlRequest -> ...
  UrlChanged url -> (AppState {s | url = url}, Cmd.None)
```

ROUNDING OUT THE VIEW

Now our application's internal logic is all set up. All that's left is the view! First let's write a couple helper functions. The first of these will parse our URL into a page so we know where we are. The second will create a link element in our page:

```
type Page =
  Contents |
  Intro |
  Conclusion |
  Other

parseUrlToPage : Url -> Page
parseUrlToPage url =
  let urlString = toString url
  in if contains "/contents" urlString
       then Contents
       else if contains "/intro" urlString
         then Intro
         else if contains "/conclusion" urlString
           then Conclusion
           else Other



link : String -> Html AppMessage
link path = a [href path] [text path]
```

Finally let's fill in a view function by applying these:

```
appView : AppState -> Document AppMessage
appView (AppState st) =
  let body = case parseUrlToPage st.url of
               Contents -> div []
                 [ link "/intro", br [] [], link "/conclusion" ]
               Intro -> div []
                 [ link "/contents", br [] [], link "/conclusion" ]
               Conclusion -> div []
```

```
            [ link "/intro", br [] [], link "/contents" ]
          Other -> div [] [ text "The page doesn't exist!" ]
in  Document "Navigation Example App" [body]
```

And now we can navigate back and forth among these pages with the links!

CONCLUSION

In this last part of our series, we completed the development of our Elm skills. We learned how to use an application to achieve the full power of a web app and navigate between different pages. There's plenty more depth we can get into with designing an Elm application. For instance, how do you structure your message types across your different pages? What kind of state do you use to manage your user's experience. These are interesting questions to explore as you become a better web developer.

And you'll also want to make sure your backend skills are up to snuff as well! Read our Haskell Web Series for more details on that! You can also download our Production Checklist!

APPENDIX: IMPORTS

```
import Browser exposing (application, UrlRequest(..), Document)
import Browser.Navigation exposing (Key, load, pushUrl)
import Html exposing (button, div, text, a, Html, br)
import Html.Attributes exposing (href)
import Html.Events exposing (onClick)
import String exposing (contains)
import Url exposing (Url, toString)
```

Revision #1
Created 11 March 2022 16:46:52 by gasick
Updated 11 March 2022 17:11:16 by gasick