

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

Elm: Functional Frontend

Frontend web programming presents an interesting challenge for Haskell developers. Ultimately, browsers run on HTML, CSS, and Javascript. This means that whatever you program in, you'd better be able to compile it down to those languages. There are a few pure Haskell approaches out there. For instance, Yesod uses Template Haskell to make HTML splices where you can substitute variables. Reflex FRP uses GHCJS to compile Haskell to Javascript.

But there are other options as well. In this series, we explore the Elm language. It has a very similar syntax to Haskell and shares many of the same functional principles. We'll get a lot of the type safety and purity we want from Haskell, in a package that compiles more easily to Javascript.

- [Elm Part 1: Language Basics](#)
- [Elm Part 2: Making a Single Page App](#)
- [Elm Part 3: Adding Effects](#)
- [Elm Part 4: Navigation](#)

Elm Part 1: Language Basics

Haskell has a number of interesting libraries for frontend web development. Yesod and Snap come to mind. Another option is Reflex FRP which uses GHCJS under the hood.

Each of these has their own strengths and weaknesses. But there are other options as well if we want to write frontend web code while keeping a functional style. This series is all about the Elm language!

I love Elm for a few reasons. Elm builds on my strong belief that we can take the principles of functional programming and put them to practical use. The language is no-nonsense and the documentation is quite good. Elm has a few syntactic quirks. It also lacks a few key Haskell features. And yet, we can still do a lot with it.

In this first part, we'll look at basic installation, and usage, as well as some differences from Haskell. If you're already a little familiar with Elm, you can move onto part 2, where we compose a simple Todo list application in Elm. This will give us a feel for how we architect our Elm applications. We'll wrap up by exploring how to add more effects to our app, and how to integrate Elm types with Haskell.

Frontend is, of course, only part of the story. To learn more about using Haskell for backend web, check out our Haskell Web Series! You can also download our Production Checklist for more ideas!

Also, be sure to check out our Github repository to see some of this example Elm code! This part's code is mostly under the ElmProject folder.

BASIC SETUP

As with any language, there will be some setup involved in getting Elm onto our machine for the first time. For Windows and Mac, you can run the installer program provided here. There are separate instructions for Linux, but they're straightforward enough. You fetch the binary, tar it, and move to your bin.

Once we have the elm executable installed, we can get going. When you've used enough package management programs, the process gets easier to understand. The elm command has a few fundamental things in common with stack and npm.

First, we can run `elm init` to create a new project. This will make a `src` folder for us as well as an `elm.json` file. This JSON file is comparable to a `.cabal` file or `package.json` for Node.js. It's where we'll specify all our different package dependencies. The default version of this will provide most of your basic web packages. Then we'll make our `.elm` source files in `/src`.

RUNNING A BASIC PAGE Elm development looks different from most normal Javascript systems I've worked with. While we're writing our code, we don't need to specify a specific entry point to our application. Every file we make is a potential web page we can view. So here's how we can start off with the simplest possible application:

```
import Browser
import HTML exposing (Html, div, text)

type Message = Message

main : Program () Int Message
main =
  Browser.sandbox { init = 0, update = update, view = view }

update : Message -> Int -> Int
update _ x = x

view : Int -> Html Message
view _ = div [] [text "Hello World!"]
```

Elm uses a model/view/controller system. We define our program in the `main` function. Our `Program` type has three parameters. The first relates to flags we can pass to our program. We'll ignore those for now. The second is the model type for our program. We'll start with a simple integer. Then the final type is a message. Our view will cause updates by sending messages of this type. The `sandbox` function means our program is simple, and has no side effects. Aside from passing an initial state, we also pass an update function and a view function.

The update function allows us to take a new message and change our model if necessary. Then the view is a function that takes our model and determines the HTML components. You can read the

type of view as "an HTML component that sends messages of type Message.

We can run the `elm-reactor` command and point our browser at `localhost:8000`. This takes us to a dashboard where we can examine any file we want. We'll only want to look at the ones with a `main` function. Then we'll see our simple page with the `div` on the screen. (It strangely spins if we select a pure library file).

As per the Elm tutorial we can make this more interesting by using the `Int` in our model. We'll change our `Message` type so that it can either represent an `Increment` or a `Decrement`. Then our `update` function will change the model based on the message.

```
type Message = Increment | Decrement

update : Message -> Int -> Int
update msg model = case msg of
  Increment -> model + 1
  Decrement -> model - 1

view : Int -> Html Message
view model = div [] [String.fromInt model]
```

As a last change, we'll add `+` and `-` buttons to our interface. These will allow us to send the `Increment` and `Decrement` messages to our type.

```
view model = div []
  [ button [onClick Decrement] [text "-"]
  , div [] [ text (String.fromInt model) ]
  , button [onClick Increment] [text "+"]
  ]
```

Now we have an interface where we can press each button and the number on the screen will change! That's our basic application!

THE MAKE COMMAND

The elm reactor command builds up a dummy interface for us to use and examine our pages. But our ultimate goal is to make it so we can generate HTML and Javascript from our elm code. We would then export these assets so our back-end could serve them as resources. We can do this with the elm make command. Here's a sample:

```
elm make Main.elm --output=main.html
```

We'll want to use scripting to pull all these elements together and dump them in an assets folder. We'll get some experience with this in a couple weeks when we put together a full Elm + Haskell project.

DIFFERENCES FROM HASKELL

There are a few syntactic gotchas when comparing Elm to Haskell. We won't cover them all, but here are the basics.

We can already see that import and module syntax is a little different. We use the exposing keyword in an import definition to pick out specific expressions we want from that module.

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

```
import Types exposing (Message(..))
```

When we define our own module, we will also use the exposing keyword in place of where in the module definition:

```
module Types exposing  
  (Message(..))
```

```
type Message = Increment | Decrement
```

We can also see that Elm uses type where we would use data in Haskell. If we want a type synonym, Elm offers the type alias combination:

```
type alias Count = Int
```

As you can see from the type operators above, Elm reverses the `:` operator and `::`. A single colon refers to a type signature. Double colons refer to list appending:

```
myNumber : Int
myNumber = 5

myList : [Int]
myList = 5 :: [2, 3]
```

Elm is also missing some of the nicer syntax elements of Haskell. For instance, Elm lacks pattern matching on functions and guards. Elm also does not have where clauses. Only case and let statements exist. And instead of the `.` operator for function composition, you would use `<<.` `data-preserve-html-node="true"` Here are a few examples of these points:

```
isBigNumber : Int -> Bool
isBigNumber x = let forComparison = 5 in x > forComparison

findSmallNumbers : List Int -> List Int
findSmallNumbers numbers = List.filter (not << isBigNumber) numbers
```

As a last note in this section, Elm is strictly evaluated. Elm compiles to Javascript so it can run in browsers. And it's much easier to generate sensible Javascript with a strict language.

ELM RECORDS

Another key difference with Elm is how record syntax works. In Elm, a "record" is a specific type. These simulate Javascript objects. In this example, we define a type synonym for a record. While we don't have pattern matching in general, we can use pattern matching on records:

```
type alias Point2D =
  { x: Float
  , y: Float
  }
```

```
sumOfPoint : Point2D -> Float
sumOfPoint {x, y} = x + y
```

To make our code feel more like Javascript, we can use the `.` operator to access records in different ways. We can either use the Javascript like syntax, or use the period and our field name as a normal function.

```
point1 : Point2D
point1 = {x = 5.0, y = 6.0}

p1x : Float
p1x = point1.x

p1y : Float
p1y = .y point1
```

We can also update particular fields of records with ease. This approach scales well to many fields:

```
newPoint : Point2D
newPoint = { point1 | y = 3.0 }
```

TYPECLASSES AND MONADS

The more controversial differences between Haskell and Elm lie with these two concepts. Elm does not have typeclasses. For a Haskell veteran such as myself, this is a big restriction. Because of this, Elm also lacks `do` syntax. Remember that `do` syntax relies upon the idea that the `Monad` typeclass exists.

There is a reason for these omissions though. The Elm creator wrote an interesting article about it.

His main point is that (unlike me), most Elm users are coming from Javascript rather than Haskell. They tend not to have much background with functional programming and related concepts. So it's not as big a priority for Elm to capture these constructs. So what alternatives are available?

Well when it comes to typeclasses, each type has to come up with its own definition for a function. Let's take the simple example of `map`. In Haskell, we have the `fmap` function. It allows us to apply a

function over a container, without knowing what the container is:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

We can apply this same function whether we have a list or a dictionary. In Elm though, each library has its own map function. So we have to qualify the usage of it:

```
import List
import Dict

double : List Int -> List Int
double l = List.map (* 2) l

doubleDict : Dict String Int -> Dict String Int
doubleDict d = Dict.map (* 2) d
```

Instead of monads, Elm uses a function called `andThen`. This acts a lot like Haskell's `>>=` operator. We see this pattern more often in object oriented languages like Java. As an example from the documentation, we can see how this works with `Maybe`.

```
toInt : String -> Maybe Int

toValidMonth : Int -> Maybe Int
toValidMonth month =
  if month >= 1 && month <= 12
  then Just month
  else Nothing

toMonth : String -> Maybe Int
toMonth rawString =
  toInt rawString `andThen` toValidMonth
```

So Elm doesn't give us quite as much functional power as we have in Haskell. That said, Elm is a front-end language first. It expresses how to display our data and how we bring components together. If we need complex functional elements, we can use Haskell and put that on the back-end.

CONCLUSION

You're now ready to move onto part 2 of this series! There, we'll expand our understanding of Elm by writing a more complicated program. We'll write a simple Todo list application and see Elm's architecture in action.

To hear more from Monday Morning Haskell, make sure to [Subscribe](#) to our newsletter! That will also give you access to our awesome [Resources](#) page!

Elm Part 2: Making a Single Page App

Welcome to part 2 of our series on Elm! Elm is a functional language you can use for front-end web development. As we explored, it lacks a few key language features of Haskell, but has very similar syntax. Checkout part 1 of this series for a refresher on that!

In this part, we're going to make a simple Todo List application to show a bit more about how Elm works. We'll see how to apply the basics we learned, and take things a bit further. If you're confident in your knowledge of the Elm architecture, you can move onto part 3, where we'll incorporate effects into our application!

But a front-end isn't much use without a back-end! Take a look at our Haskell Web Series to learn some cool libraries for a Haskell back-end!

As an extra note, all the code for this series is on Github. The code for this section is ElmTodo directory!

TODO TYPES

Before we get started, let's define our types. We'll have a basic Todo type, with a string for its name. We'll also make a type for the state of our form. This includes a list of our items as well as a "Todo in Progress", containing the text in the form:

```
module Types exposing
  ( Todo(..)
  , TodoListState(..)
  , TodoListMessage(..)
  )

type Todo = Todo
```

```
{ todoName : String }

type TodoListState = TodoListState
  { todoList : List Todo
    , newTodo : Maybe Todo
  }
```

We also want to define a message type. These are the messages we'll send from our view to update our model.

```
type TodoListMessage =
  AddedTodo Todo |
  FinishedTodo Todo |
  UpdatedNewTodo (Maybe Todo)
```

ELM'S ARCHITECTURE

Now let's review how Elm's architecture works. Last week we described our program using the `sandbox` function. This simple function takes three inputs. It took an initial state (we were using a basic `Int`), an update function, and a view function. The update function took a `Message` and our existing model and returned the updated model. The view function took our model and rendered it in HTML. The resulting type of the view was `Html Message`. You should read this type as, "rendered HTML that can send messages of type `Message`". The resulting type of this expression is a `Program`, parameterized by our model and message type.

```
sandbox :
  { init : model
    , update : msg -> model -> model
    , view : model -> Html msg
  }
  -> Program () model msg
```

A sandbox program though doesn't allow us to communicate with the outside world very much! In other words, there's no IO, except for rendering the DOM! So there are a few more advanced functions we can use to create a `Program`. For a normal application, you'll want to use the application function seen here. For the single page example we'll do this week, we can pretty much get away

with sandbox. But we'll show how to use the element function instead to get at least some effects into our system. The element function looks a lot like sandbox, with a few changes:

```
element :  
  { init : flags -> (model, Cmd msg)  
  , view : model -> Html msg  
  , update : msg -> model -> (model, Cmd msg)  
  , subscriptions : model -> Sub msg  
  }  
  -> Program flags model msg
```

Once again, we have functions for init, view, and update. But a couple signatures are a little different. Our init function now takes program flags. We won't use these. But they allow you to embed your Elm project within a larger Javascript project. The flags are information passed from Javascript into your Elm program.

Using init also produces both a model and a Cmd element. This would allow us to run "commands" when initializing our application. You can think of these commands as side effects, and they can also produce our message type.

Another change we see is that the update function can also produce commands as well as the new model. Finally, we have this last element subscriptions. This allows us to subscribe to outside events like clock ticks and HTTP requests. We'll see more of this next week. For now, let's lay out the skeleton of our application and get all the type signatures down. (See the appendix for an imports list).

```
main : Program () TodoListState TodoListMessage  
main = Browser.element  
  { init = todoInit  
  , update = todoUpdate  
  , view = todoView  
  , subscriptions = todoSubscriptions  
  }  
  
todoInit : () -> (TodoListState, Cmd TodoListMessage)  
  
todoUpdate : TodoListMessage -> TodoListState -> (TodoListState, Cmd TodoListMessage)
```

```
todoView : TodoListState -> Html TodoListMessage
```

```
todoSubscriptions : TodoListState -> Sub TodoListMessage
```

Initializing our program is easy enough. We'll ignore the flags and return a state that has no tasks and Nothing for the task in progress. We'll return Cmd.none, indicating that initializing our state has no effects. We'll also fill in Sub.none for the subscriptions.

```
todoInit : () -> (TodoListState, Cmd TodoListMessage)
todoInit _ =
  let st = TodoListState { todoList = [], newTodo = Nothing }
  in (st, Cmd.none)

todoSubscriptions : TodoListState -> Sub TodoListMessage
todoSubscriptions _ = Sub.none
```

FILLING IN THE VIEW

Now for our view, we'll take our basic model components and turn them into HTML. When we have a list of Todo elements, we'll display them in an ordered list. We'll have a list item for each of them. This item will state the name of the item and give a "Done" button. Clicking the button allows us to send a message for finishing that Todo:

```
todoItem : Todo -> Html TodoListMessage
todoItem (Todo todo) = li []
  [ text todo.todoName
  , button [onClick (FinishedTodo (Todo todo))] [text "Done"]
  ]
```

Now let's put together the input form for adding a Todo. First, we'll determine what value is in the input and whether to disable the done button. Then we'll define a function for turning the input string into a new Todo item. This will send the message for changing the new Todo.

```
todoForm : Maybe Todo -> Html TodoListMessage
todoForm maybeTodo =
  let (value_, isEnabled_) = case maybeTodo of
```

```

        Nothing -> ("", False)
        Just (Todo t) -> (t.todoName, True)

changeTodo newString = case newString of
    "" -> UpdatedNewTodo Nothing
    s -> UpdatedNewTodo (Just (Todo { todoName = s }))

in ...

```

Now, we'll make the HTML for the form. The input element itself will tie into our onChange function that will update our state. The "Add" button will send the message for adding the new Todo.

```

todoForm : Maybe Todo -> Html TodoListMessage
todoForm maybeTodo =
    let (value_, isEnabled_) = ...
        changeTodo newString = ...
    in div []
        [ input [value value_, onInput changeTodo] []
        , button [disabled (not isEnabled_), onClick (AddedTodo (Todo {todoName = value_}))]
            [text "Add"]
        ]

```

We can then pull together all our view code in the view function. We have our list of Todos, and then add the form.

```

todoView : TodoListState -> Html TodoListMessage
todoView (TodoListState { todoList, newTodo }) = div []
    [ ol [] (List.map todoItem todoList)
    , todoForm newTodo
    ]

```

UPDATING THE MODEL

The last thing we need is to write out our update function. All this does is process a message and update the state accordingly. We need three cases:

```

todoUpdate : TodoListMessage -> TodoListState -> (TodoListState, Cmd TodoListMessage)
todoUpdate msg (TodoListState { todoList, newTodo }) = case msg of
    (AddedTodo newTodo_) -> ...

```

```
(FinishedTodo doneTodo) -> ...  
(UpdatedNewTodo newTodo_) -> ...
```

And each of these cases is pretty straightforward. For adding a Todo, we'll append it at the front of our list:

```
todoUpdate : TodoListMessage -> TodoListState -> (TodoListState, Cmd TodoListMessage)  
todoUpdate msg (TodoListState { todoList, newTodo}) = case msg of  
  (AddedTodo newTodo_) ->  
    let st = TodoListState { todoList = newTodo_ :: todoList  
                          , newTodo = Nothing  
                          }  
  (FinishedTodo doneTodo) -> ...  
  (UpdatedNewTodo newTodo_) -> ...
```

When we've finished a Todo, we'll remove it from our list by filtering on the name being equal.

```
todoUpdate : TodoListMessage -> TodoListState -> (TodoListState, Cmd TodoListMessage)  
todoUpdate msg (TodoListState { todoList, newTodo}) = case msg of  
  (AddedTodo newTodo_) -> ...  
  (FinishedTodo doneTodo) ->  
    let st = TodoListState { todoList = List.filter (todosNotEqual doneTodo) todoList  
                          , newTodo = newTodo  
                          }  
    in (st, Cmd.none)  
  (UpdatedNewTodo newTodo_) -> ..  
  
todosNotEqual : Todo -> Todo -> Bool  
todosNotEqual (Todo t1) (Todo t2) = t1.todoName /= t2.todoName
```

And updating the new todo is the easiest of all! All we need to do is replace it in the state.

```
todoUpdate : TodoListMessage -> TodoListState -> (TodoListState, Cmd TodoListMessage)  
todoUpdate msg (TodoListState { todoList, newTodo}) = case msg of  
  (AddedTodo newTodo_) -> ...  
  (FinishedTodo doneTodo) -> ...  
  (UpdatedNewTodo newTodo_) -> ..  
    (TodoListState { todoList = todoList, newTodo = newTodo_ }, Cmd.none)
```

And with that we're done! We have a rudimentary program for our Todo list.

CONCLUSION

This wraps up our basic Todo application! You're now ready for part 3 of this series! We'll see how Elm effect system works, and use it to send HTTP requests.

For some more ideas on building cool products in Haskell, take a look at our [Production Checklist](#). It goes over some libraries for many topics, including databases and parsing!

APPENDIX: IMPORTS

```
import Browser
import Html exposing (Html, button, div, text, ol, li, input)
import Html.Attributes exposing (value, disabled)
import Html.Events exposing (onClick, onInput)
```


Elm Part 3: Adding Effects

In part 2 of this series, we dug deeper into using Elm. We saw how to build a more complicated web page for a Todo list application. We learned about the Elm architecture and saw how we could use a couple simple functions to build our page. We laid the groundwork for bringing effects into our system, but didn't use any of these.

This week, we'll add some useful pieces to our Elm skill set. We'll see how to include more effects in our system, specifically randomness and HTTP requests.

To learn more about constructing a backend for your system, you should read up on our Haskell Web Series. It'll teach you things like connecting to a database and making an HTTP server.

Once you're done with this article, you'll be ready for the fourth and final part of this series. We'll cover the basics of Navigation for a multi-page application. As a reminder, you can also look at all the code for this series on Github! This section's code is in the ElmTodo folder.

INCORPORATING EFFECTS

Last week, we explored using the `element` expression to build our application. Unlike `sandbox`, this allowed us to add commands, which enable side effects. But we didn't use any of commands. Let's examine a couple different effects we can use in our application.

One simple effect we can cause is to get a random number. It might not be obvious from the code we have so far, but we can't actually do it in our Todo application at the moment! Our `update` function is pure! This means it doesn't have access to IO. What it can do is send commands as part of its output. Commands can trigger messages, and incorporate effects along the way.

MAKING A RANDOM TASK

We're going to add a button to our application. This button will generate a random task name and add it to our list. To start with, we'll add a new message type to process:

```
type TodoListMessage =  
  AddedTodo Todo |  
  FinishedTodo Todo |  
  UpdatedNewTodo (Maybe Todo) |  
  AddRandomTodo
```

Now here's the HTML element that will send the new message. We can add it to the list of elements in our view:

```
randomTaskButton : Html TodoListMessage  
randomTaskButton = button [onClick AddRandomTodo] [text "Random"]
```

Now we need to add our new message to our update function. We need a case for it:

```
todoUpdate : TodoListMessage -> TodoListState -> (TodoListState, Cmd TodoListMessage)  
todoUpdate msg (TodoListState { todoList, newTodo }) = case msg of  
  ...  
  AddRandomTodo ->  
    (TodoListState { todoList = todoList, newTodo = newTodo }, ...)
```

So for the first time, we're going to fill in the Cmd element! To generate randomness, we need the generate function from the Random module.

```
generate : (a -> msg) -> Generator a -> Cmd msg
```

We need two arguments to use this. The second argument is a random generator on a particular type a. The first argument then is a function from this type to our message. In our case, we'll want to generate a String. We'll use some functionality from the package `elm-community/random-extra`. See `Random.String` and `Random.Char` for details. Our strings will be 10 letters long and use only lowercase.

```
genString : Generator String  
genString = string 10 lowerCaseLatin
```

Now we can easily convert this to a new message. We generate the string, and then add it as a `Todo`:

```
addTaskMsg : String -> TodoListMessage
addTaskMsg name = AddedTodo (Todo {todoName = name})
```

Now we can plug these into our update function, and we have our functioning random command!

```
todoUpdate : TodoListMessage -> TodoListState -> (TodoListState, Cmd TodoListMessage)
todoUpdate msg (TodoListState { todoList, newTodo}) = case msg of
  ...
  AddRandomTodo ->
    (... , generate addTaskMsg genString)
```

Now clicking the random button will make a random task and add it to our list!

SENDING AN HTTP REQUEST

A more complicated effect we can add is to send an HTTP request. We'll be using the `Http` library from Elm. Whenever we complete a task, we'll send a request to some endpoint that contains the task's name within its body.

We'll hook into our current action for `FinishedTodo`. Currently, this returns the `None` command along with its update. We'll make it send a command that will trigger a post request. This post request will, in turn, hook into another message type we'll make for handling the response.

```
todoUpdate : TodoListMessage -> TodoListState -> (TodoListState, Cmd TodoListMessage)
todoUpdate msg (TodoListState { todoList, newTodo}) = case msg of
  ...
  (FinishedTodo doneTodo) ->
    (... , postFinishedTodo doneTodo)
  ReceivedFinishedResponse -> ...

postFinishedTodo : Todo -> Cmd TodoListMessage
postFinishedTodo = ...
```

We create HTTP commands using the `send` function. It takes two parameters:

```
send : (Result Error a -> msg) -> Request a -> Cmd Msg
```

The first of these is a function interpreting the server response and giving us a new message to send. The second is a request expecting a result of some type `a`. Let's plot out our code skeleton a little more for these parameters. We'll imagine we're getting back a `String` for our response, but it doesn't matter. We'll send the same message regardless:

```
postFinishedTodo : Todo -> Cmd TodoListMessage
postFinishedTodo todo = send interpretResponse (postRequest todo)

interpretResponse : Result Error String -> TodoListMessage
interpretResponse _ = ReceivedFinishedResponse

postRequest : Todo -> Request String
postRequest = ...
```

Now all we need is to create our post request using the `post` function:

```
post : String -> Body -> Decoder a -> Request a
```

Now we've got three more parameters to fill in. The first of these is the URL we're sending the request to. The second is our body. The third is a decoder for the response. Our decoder will be `Json.Decode.string`, a library function. We'll imagine we are running a local server for the URL.

```
postRequest : Todo -> Request String
postRequest todo = post "localhost:8081/api/finish" ... Json.Decode.string
```

All we need to do now is encode the `Todo` in the post request body. This is straightforward. We'll use the `Json.Encode.object` function, which takes a list of tuples. Then we'll use the string encoder on the todo name.

```
jsonEncTodo : Todo -> Value
jsonEncTodo (Todo todo) = Json.Encode.object
  [ ("todoName", Json.Encode.string todo.todoName) ]
```

We'll use it together with the `jsonBody` function. And then we're done!

```
postRequest : Todo -> Request String
postRequest todo = post
  "localhost:8081/api/finish"
  (jsonBody (jsonEncTodo todo))
  Json.Decode.string
```

As a reminder, most of the types and helper functions from this last part come from the HTTP Library for Elm. We could then further process the response in our `interpretResponse` function. If we get an error, we could send a different message. Either way, we can ultimately do more updates in our update function.

CONCLUSION

This concludes part 3 of our series on Elm! We took a look at a few nifty ways to add extra effects to our Elm projects. We saw how to introduce randomness into our Elm project, and then how to send HTTP requests. In part 4, we'll wrap up our series by looking at navigation, a vital part of any web application. We'll see how the Elm architecture supports a multi-page application. Then we'll see how to move between the different pages efficiently, without needing to reload every bit of our Elm code each time.

Now that you know how to write a functional frontend, you should learn more about the backend! Read our Haskell Web Series for some tutorials on how to do this. You can also download our Production Checklist for some more ideas!

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)
```