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

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 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)
```

Revision #1
Created 2022-03-11 16:41:06 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick