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

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
interpretResposne _ = 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!

Revision #1
Created 11 March 2022 16:43:51 by gasick
Updated 11 March 2022 17:11:16 by gasick