Elm

Introdução e comparação com React/Redux

Rodrigo Stevaux @ Jaya Labs

March 6, 2018

Elm: linguagem + arquitetura

Linguagem

- Funcional pura
- Tipos estáticos c/ inferência
- Similar a F#, Haskell e OCaml
- Especifica para SPA

Arquitetura

- Modelo: estado da aplicação
- Update: recebe comandos e atualiza modelo
- View: renderiza modelo em um DOM
- Equivalente a Redux + React + Thunk ou Saga

Resultado: código fácil de manter e estender, clientes e devs mais felizes, menos bugs

Arquitetura básica

Modelos

O estado completo da aplicação é representado por um modelo, que geralmente é um record/struct com vários campos, e tudo é tipado:

```
type alias Model = {
  running : Bool,
  elapsed : Float,
  pastLaps : List Float
}
```

Em um cronômetro, o estado é se o relógio está parado ou correndo, qual o tempo acumulado na volta atual, e quais os tempos das voltas passadas.

Mensagens

Mensagens carregam comandos para o aplicativo fazer qualquer coisa útil: alterar alguma variável do estado, fazer algum request para falar com alguma API, aumentar um counter, etc.

```
type Msg =
    Start
  | Stop
  | Reset
  | Lap
  | Tick Time
```

Update

Atualizamos o estado da aplicação enviando `Msg` para o `update`, que transforma o `model` anterior e retorna um novo:

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Start -> { model | running = True }
    Stop  -> { model | running = False }
    Reset -> { model | elapsed = 0 }
    -- h :: t coloca o elemento h na frente da lista t
    Lap   -> { model | laps = model.elapsed :: model.laps,
                       elapsed = 0 }
    Tick time -> { model | elapsed = Time }
```

Isso é bom, pois em um lugar só podemos ver tudo que pode acontecer com o modelo, e, com os tipos, o Elm só compila a aplicação se todas as mensagens forem tratadas!

Views

As views no Elm são apenas funções que geram HTML a partir do modelo (Os equivalentes no React seriam componentes puros.)

```
view model = div [] [
  div [] [h1 [] [text <| format "%M:%H:%S" model.elapsed]],
  , div [] [b [] [text "Laps:"]]
  , div [] (List.map viewLap model.laps)
  , div [] [button [onClick Start] [text "Start"]]
  , div [] [button [onClick Stop] [text "Stop"]]
]
viewLap lap = div [] [text <| format "%M:%H:%S" model.elapsed]
```

Esse `onClick Start` faz uma mensagem `Start` chegar ao `update`.

Subscriptions

O Elm tem um conceito de "subscriptions" para ouvir eventos do browser como timers, mouse e keyboard, por exemplo. Conforme o modelo mudar, as subsriptions tambem podem mudar, como ilustrado na função seguinte:

```
subscriptions : Model -> Sub Msg
subscriptions model =
  if model.running then
    Time.every (20 * millisecond) Tick
  else
    Sub.none
```

Cada subscription envia uma `Msg` para o `update`. Se o cronometro está ligado, queremos ser avisados com a mensagem `Tick` no `update` a cada 20 ms

E como juntar as partes?

A aplicação tem uma função main, que usa a função program, para juntar as partes da arquitetura:

```
main =
  program init view update subscriptions

init = ..., view = ...
update = ..., subscriptions = ...
```

Isto tudo é compilado para um único .JS para incluirmos em um HTML. Elm não precisa ser usado na página toda.

```
<script src="app.js"/>
<div id="root"></div>
<script>
  var root = document.getElementById("root");
  var app = Elm.Main.embed(root);
</script>
```

APIs: usando HTTP e JSON

JSON: transformando em valores do Elm

Decode de JSON é representado de forma type-safe:

```
type Decoder a = Json.Value -> a

decodeValue : Decoder a -> Value -> Result String a
```

Ou seja: para decodificar um valor JSON para um valor Elm nós aplicamos um Decoder a uma String, e temos como resultado ou uma mensagem de erro ou um valor do tipo a

Exemplo

Tem muita coisa funcional acontecendo atrás disso, mas para os usuários basta usar como uma DSL:

```
import Decode as D

type alias Model = { elapsed, running, laps }

decodeModel : D.Decoder Model
decodeModel =
    D.map2 Todo
        (D.field "elapsed" D.float)
        (D.field "running" D.bool)
        (D.field "laps" (D.list D.string))
```

HTTP

As chamadas HTTP no Elm são assíncronas e se encaixam na arquitetura através do `update`:

- Uma chamada HTTP é um `Cmd Msg`, que é executado pelo run-time do Elm
- Quando chegar a resposta, o run-time envia uma `Msg` para `update`

A chamada é organizada em 3 partes:

1. A url e o verbo HTTP
2. Um decoder para a resposta com tipo `Decoder a`
3. Um caso de `Msg` com um campo `Result Http.Error a`

Exemplo

```
getAppState : Cmd Msg
getAppState =
  let req = Http.get "api.webpage.com/states/" decodeModel
  in Http.send req ModelFromServer

type Msg =
  ...
  | ModelFromServer (Result Http.Error Model)

update msg model =
  case msg of
    ...
    ClickRestore -> (model, getAppState)
    ModelFromServer (Ok mod) -> (mod, Cmd.none)
    ModelFromServer (Err err) -> ...
```

Demo

Por quê usar Elm se parece tanto com React?

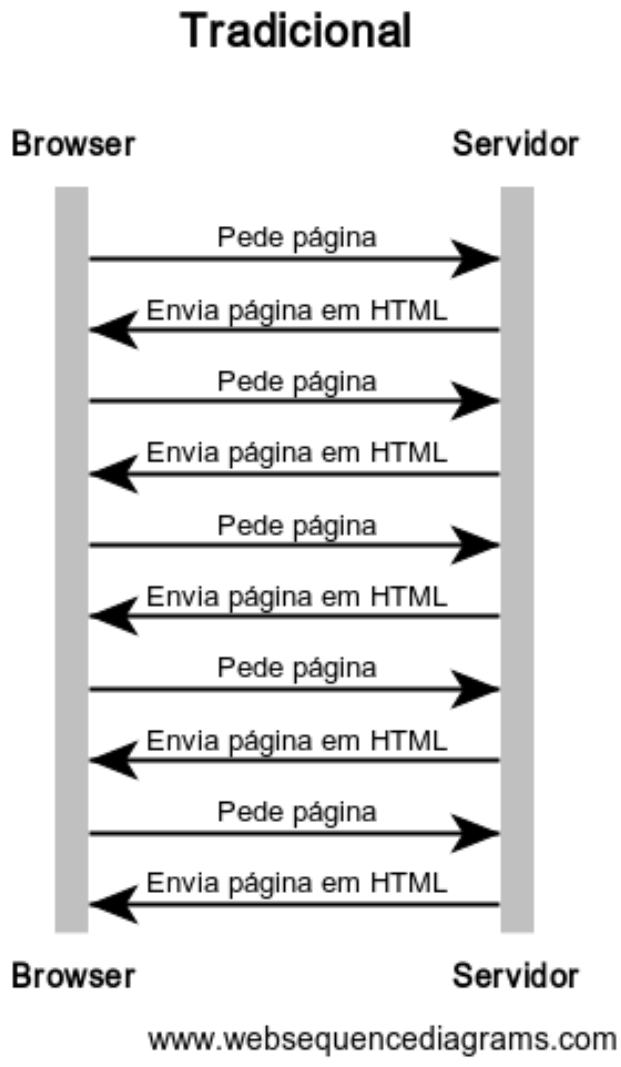
Elm gera software mais correto, robusto e gera menos stress para desenvolvedores e clientes.

- Os tipos deixam tudo mais seguro, checado pelo compilador
- O compilador é um guia para o processo de desenvolvimento
- Refatorar é muito mais fácil com tantos checks no compilador
- Async done right
- Build system super simples - um só comando

O jeito mais fácil é demonstrar isso em uma sessão de live coding

Backup

Tradicional



Single-page applications