Elm |> Real Life

jiri.sliva@newired.com

Agenda

Jak jsme strukturovali kód

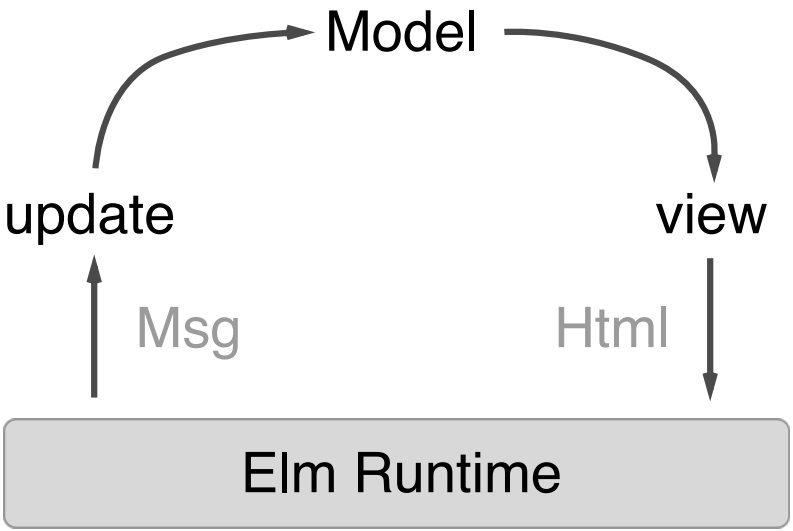
Jaké problémy jsme museli řešit

Jaké jsou nedostatky

Co se nám na Elmu líbí

Jak členit velkou Elm aplikaci na menší části

- Model - desítky typů
- Messages + Update funkce - desítky message typů a Update funkcí
- Views - desítky funkcí



1. Maximálně odělit části aplikace

Každá "komponenta" v jednom Elm modulu s vlastním

- Modelem
- Messages + update funkce
- Views funkce

```elm
-- Login.elm

type alias Model =
    { username : String
    , password : String
    }

type Msg
    = EnterName String
    | EnterPass String

update : Msg -> Model -> Model
..

view : Model -> Html Msg
..
```

```elm
-- Project.elm

type alias Model =
    { name : String
    , active : Bool
    }

type Msg
    = SetName String
    | Activate Bool

update : Msg -> Model -> Model
..

view : Model -> Html Msg
..
```

- Model se skládá z dílčích sub-modelů komponent.
- update funkce převolává update funkce komponent.
- Messages komponent se mapují na Msg typ z Main.elm.

```
-- Main.elm

import Login
import Project

type alias Model =
    { login : Login.Model
    , projects : List Project.Model
    }

type Msg
    = LoginMsg Login.Msg
    | ProjectMsg Project.Msg

update : Msg -> Model -> Model
update msg model =
  case msg of
    LoginMsg loginMsg ->
      { model | login = Login.update loginMsg model.login }
    ..


view : Model -> Html Msg   <-------------------------+
view model =                                         | different Msg types
  ..                                                 |
  Html.map LoginMsg (Login.view model.login)  <------+
```

Přínosy

- Kód je lépe znovupoužitený.
- Kód je nezávislý na ostatních částech aplikace.

Nevýhody

- Mapování zpráv a updatů z "parenta" na "child"
- Update "parent" modelu z "child" update funkce

2. "As flat as possible"

```elm
-- Model.elm

type alias Login =
    { username : String
    , password : String
    }

type alias Project =
    { name : String
    , active : Bool
    }

type alias Model =
    { login : Login
    , projects : List Projects
    }
```

```elm
-- Msg.elm

type Msg
   = LoginMsg LoginMsg
   | ProjectMsg ProjectMsg

type LoginMsg
   = EnterName String
   | EnterPass String

type ProjectMsg
   = SetName String
   | Activate Bool
```

```elm
-- View/Login.elm

import Msg exposing (Msg(..))
import Model exposing (Login)

view : Login -> Html Msg
..
```

```elm
-- View/Project.elm

import Msg exposing (Msg(..))
import Model exposing (Project)

view : Project -> Html Msg
..
```

```elm
-- Update.elm

update : Msg -> Model -> Model
update msg model =
  case msg of
    LoginMsg loginMsg ->
        { model | login = (LoginUpdate.update loginMsg model.login) }
    ..
```

Přínosy

- Jednodušší kód - žádné mapování
- Odpadá problém updatů Child -> Parent

Nevýhody

- Kompaktní kód není bez refactoru možné použít jinde.

Závěr

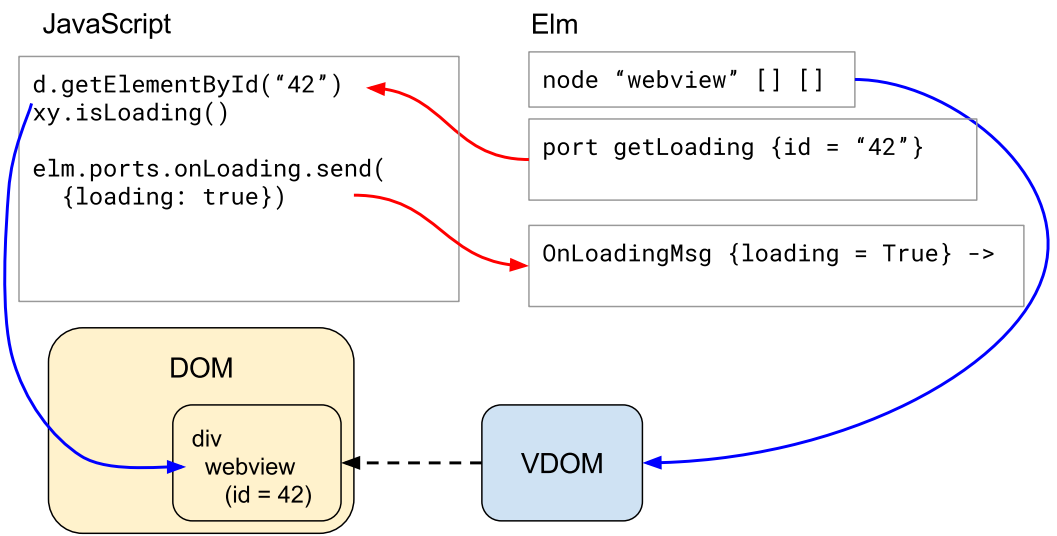
- Oddělovat typy Messages jen v případě "obecných" komponent (např. datepicker).
- Nevynucovat si OOP přístup za každou cenu. Místo snahy vytvářet znovupoužitelné *komponenty* psát znovupoužitelné *view funkce*.

Problémy, a jak jsme je vyřešili

- VDOM a omezení která s něj vyplývají.
  - Není možné manipulovat s elementy generovanými VDOMem. Změny mohou být kdykoliv ztraceny. (včetně přidávání evet listeneru)
  - Nemá smysl ukládat si reference na elementy. Po příšním renderu DOMu můžou být reference úplně jiné.
- Z Elmu není přímo dostupné JS API elementů.
  - Porty - asynchronní volání může komplikovat logiku aplikace
  - Native module - riziko zavlečení chyby a ztráta stability Elmu

Náš příklad - webview (Electron)

1. Životní cyklus webviews řešený kompletně v JavaScriptu a porty komunikovat s Elmem. Nevýhoda: Kritická část *stavu* chybí v modelu Elm aplikace.
2. Životní cyklus webviews řízený v Elmu. Pro volání JS API napsat Native module. Ani jsme to nezkoušeli. :-)
3. Životni cyklus webviews řízený v Elmu. Pro volání JS API používat porty.



Co chybí

IDE/Editor

- **Atom** - Skvělá podpora Elmu. Pomalý, *náladový* editor.
- **VS Code** - Základní podpora Elmu (velmi nedokonalé *go to definition*)
- **IntelliJ IDEA** - Základní podpora Elmu (nevolá kompilátor).
- **Sublime Text** - Slušná podpora Elmu - Vývoj editoru se pravděpobně zastavil.
- **LightTable** - Skvělá podpora Elmu - autorem opuštěný projekt (komunita se nezdá být příliš aktivní)
- **Vim** ;-)

Co chybí

Debugger

- Default - skvěká věc pro testing. Málo features pro debugging.
- `github.com/jinjor/elm-time-travel` - Lepší pro debugging (alfa).

Privátní package

Dnes jsou možné pouze veřejné balíky přes github (Elm 0.19?)

Co dělá radost - Platforma

- Rychlý vývoj. Je snadné začít s prototypem a pak refaktorovat a pridávat další věci.
- Bezproblémový refactoring
  - Nejen díky statickému typování, ale také díky jedinému *místu* kde leží stav.
- *No runtime errors* - legenda je nelže :)
- Radost! Čistě funkcionální jazyk ve spojení s *Elm Architecture* působí velmi přirozeně (Event -> Update -> View).
  - Elm může posloužit jako vstupní jazyk do funcionáního světa.
  - Dnes už bych se neobával použít čistě funkcionální jazyk i pro backend (Elixir?)
- Hodně částí, které jsme plánovali psát JavaScriptu jsme nakonec přepsali do Elmu.
- HTML to Elm - něco co vypadá jako hříčka dokáže ušetřit hodně času.

Co dělá radost - Adaptace

- Vyvojář bez předchozí zkušenosti s funkcionálními jazyky byl schopen psát produkční kód po 3 týdnech.
- Elm komunita (např. na Slacku) je *velmi* vstřícná k začátečníkům.