Obelisk - FRP

Alexandre Garcia de Oliveira

Outubro 2021

Chapter 1

Introdução

O Obelisk é uma framework (Haskell) para desenvolver projetos web de uma forma similar ao react do JavaScript. A ideia é que possamos programar HTML e JavaScript sem sair do Haskell, ou seja, a ferramenta proverá uma tradução do Haskell para essas linguagens. A framework se baseia no conceito de Programação Funcional Reativa Pura (FRP) que visa modelar programar que possuam algum tipo de interatividade. Esse paradigma visa a manipulação de eventos e seus comportamentos usando a programação funcional.

Essa ferramenta possui a vantagem de possuir um live reloader (quando modificamos um arquivo .hs temos uma atualização automática da página sem a necessidade de compilar manualmente o projeto). Além disso a ferramenta também possui módulo para compilar seus fontes visando arquiteturas mobile (iOS e Android).

A ferramenta Obelisk ela é baseada nas bibliotecas reflex (para o controle dos eventos e comportamentos) e reflex-dom (para o controle dos elementos do Document Objet Model do JavaScript) que controla toda a parte de frontend da ferramenta. As aplicações feitas podem ser de página única (single page) ou com várias páginas usando um esquema rotas (routing). As ferramantas para a manipulação do back-end (persistncia de dados) serão feitas com bibliotecas terceiras como postgresql-simple para a conexão e manipulação de um banco de dados postgresql. Um esquema rotas (routing) será feito em conjunto com o Obelisk de forma a ter um fácil acesso aos end-points definidos. O padrão de aplicação REST será seguido para termos uma aplicação exemplo básica que demonstre o link entre a framework Obelisk e um banco de dados postgresql.

O Obelisk possui sua documentação através das seguintes fontes:

- Repositório oficial (Instalação, fonte e exemplos): https://github.com/obsidiansystems/obelisk
- Sobre o reflex e o reflex-dom (documentação e exemplos): https://reflex-frp.org/
- Uma útil lista de funções do reflex-dom:

https://github.com/reflex-frp/reflex-dom/blob/develop/Quickref.md

- Uma útil lista de funções do reflex:
  https://github.com/reflex-frp/reflex/blob/develop/Quickref.md

Para o back-end, a ferramenta postgresql-simple possui o seguinte repositório: https://hackage.haskell.org/package/postgresql-simple.

1.1 Instalação

Para um desenvolvimento confortável com a ferramenta Obelisk necessita-se de um sistema operação Linux com alguma distribuição de sua preferncia (Ubuntu, Alpine, Arch, etc...). Ou para um melhor resultado recomenda-se o sistema operacional NixOS (que não será abordado neste trabalho).

O foco deste trabalho será em um ambiente Ubuntu 20.01 que tenha o gerenciador de pacotes nix (note que não é o gerenciador de pacotes padrão, o apt). O uso do nix para instalação desta ferramenta se faz obrigatório.

Alguns pré-requistos para a instalação do Obelisk em ambiente Ubuntu:

- Por causa do nix, o sistema /root deve ser montado numa partição com pelo menos 20GB.
- Partição swap padrão e uma partição /home com pelo menos 10GB.
- O gerenciador nix.
- Browser: Google Chrome.
- O programa curl (sudo apt install curl).
- O programa vim (sudo apt install vim).
- Editor de código: Kate, Sublime, etc...

A instalação deve seguir os seguintes passos conforme a documentação do Obelisk:

- instalar o nix:

  ```
  sh <(curl -L https://nixos.org/nix/install) --daemon
  ```

- Verifique se o arquivo abaixo existe

  ```
  ls /etc/nix/nix.conf
  ```

  Caso não exista, crie

  ```
  sudo touch /etc/nix/nix.conf
  ```

  Caso exista, apenas modifique

  ```
  sudo vim /etc/nix/nix.conf
  ```

e coloque as seguintes linhas (dentro do vim use para inserir :i, para salvar :w e para sair :q)

```
binary-caches = https://cache.nixos.org https://nixcache.reflex-frp.org
binary-cache-public-keys = cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY
binary-caches-parallel-connections = 40
```

- Instalar via nix

```
nix-env -f https://github.com/obsidiansystems/obelisk/archive/master.tar.gz -iA command
```

Após a instalação devemos criar um novo projeto usando o Obelisk.

```
$ mkdir projeto
$ cd projeto
$ ob init
```

o comando "ob init" serve para baixar a estrutura completa de um projeto.

1.2 Estrutura de um projeto

O projeto deverá conter as seguintes pastas:

- backend: terá os módulos Haskell para a manipulação da parte de servidores e banco de dados de um projeto;
- frontend: terá os módulos Haskell para a manipulação das páginas dinâmicas. É aqui que usaremos os pacotes reflex e reflex-dom;
- static: possui arquivos estáticos como css, js (por exemplo, bootstrap), imagens, fontes, entre outros;
- common: módulos que sejam comnuns ao backend e ao frontend, por exemplo as rotas;
- config: possui arquivos de configuração.

Nesse inicio, o módulo contido em

`/frontend/src/Frontend.hs`

será o mais importante, pois é nele que é definido a página a ser desenvolvida. Após entedermos o mecanismo que rege o reflex e o reflex-dom partiremos para o entendimento do backend que é o arquivo contido em

`/backend/src/Backend.hs`

e o esquema de rotas fica localizado no módulo

`common/src/Common/Route.hs`

Para chamar o livre-reload (projeto) do Obelisk batta executar o comando abaixo na raiz do projeto.

```
ob run
```

Para ver a página de "hello world" devemos acessar no browser a url http://localhost:8000 após a instalação de todas as bibliotecas ao qual o Obelisk depende. A mensagem "Frontend running on http://localhost:8000" indica que o processo está pronto. A instalação das dependncias ocorrerá apenas na primeira vez.

1.3 Entendendo o primeiro projeto

No arquivo Frontend.hs existe a função frontend que toma conta de toda a página principal da aplicação.

```
frontend :: Frontend (R FrontendRoute)
frontend = Frontend
  { _frontend_head = do
      el "title" $ text "Obelisk Minimal Example"
      elAttr "link" ("href" =: static @"main.css"
                     <> "type" =: "text/css"
                     <> "rel" =: "stylesheet") blank
  , _frontend_body = do
      el "h1" (text "Welcome to Obelisk!")
      el "p" $ text $ T.pack commonStuff

      elAttr "img" ("src" =: static @"obelisk.jpg") blank
      el "div" $ do
        exampleConfig <- getConfig "common/example"
        case exampleConfig of
          Nothing -> text "No config file found in config/common/example"
          Just s -> text $ T.decodeUtf8 s
      return ()
  }
```

A função frontend possui o tipo Frontend (R FrontendRoute) que é nada mais nada menos que um record syntax com dois campos _frontend_head e _frontend_body que representam as tags head e body de um html. Aqui usa-se a notação do (monad) para sequenciar os elementos da página. Note que aqui estamos usamos funções do reflex-dom. No campo _frontend_head definimos que a página terá uma estrutura parecida com as tags a seguir.

```
<head>
    <title>
         Obelisk Minimal Example
    </title>
    <link href="main.css" type="text/css" rel="stylesheet">
</head>
```

A chamada da função el no head

```
el "title" (text "Obelisk Minimal Example")
```

indica que queremos criar a tag title (string do primeiro parâmetro), a função de dentro text indica que é um nó texto subordinado ao title. A função elAttr cria tags com atributos nela conforme a linha a seguir.

```
elAttr "link" ("href" =: static @"main.css"
                    <> "type" =: "text/css"
                    <> "rel" =: "stylesheet") blank
```

Nesse caso, cria-se a tag link com os atributos href, type e rel. A escrita é usando as funções definidas para um Map (tipo do Haskell que modela um dicionário chave/valor). O operador "(:=)" indica que estamos criando um registro ao qual o lado esquerdo é a chave e o direito o valor. O operador (<>) é o mappend do monóide e serve para juntar os registros. No exemplo temos que uma chave é a string "href" e valor uma menção à pasta static com o conteúdo "main.css" (note que se esse arquivo não existir, o projeto não compilará). O blank é uma função que indica que não tags subordinadas à tag link.

Em _frontend_body define-se a estrutura para o corpo da página (body). Nota-se que na primeira temos a chamada da função el com a string h1 indicando a criação de uma tag h1 com a mensagem "Welcome to Obelisk" parãmetro da função text.

```
 el "h1" (text "Welcome to Obelisk!")
```

Na segunda linha vemos novamente a ação da função el para a criação de um parágrafo.

```
 el "p" $ text $ T.pack commonStuff
```

a função text recebe como parãmetro a expressão

```
T.pack commonStuff
```

que está convertendo para Text a String retornada pela função commonStuff definida no arquivo

```
/common/src/Common/Api.hs
```

que possui a função mencionada.

```
commonStuff :: String
commonStuff = "Here is a string defined in Common.Api"
```

Lembramos que Text e String são duas representações diferentes para modelar caractéres em Haskell.

A terceira linha do body é uma imagem que vai possuir atributos, para isso usa-se a função elAttr e não el como visto anteriormente.

```
elAttr "img" ("src" =: static @"obelisk.jpg") blank
```

Está imagem vai ser procurada na pasta static e ela não possui nenhuma tag subordinada. Vale lembrar que devemos reiniciar o live reloader para que esta busca seja feita.

Finalmente, a última linha indica que queremos mostrar, dentro de uma div, uma mensagem dependendo da existncia de um arquivo na pasta

```
config/common/example
```

Se ele existir, seu conteúdo será exibido, caso contrário, mostrará uma mensagem de erro.

```
el "div" $ do
        exampleConfig <- getConfig "common/example"
        case exampleConfig of
          Nothing -> text "No config file found in config/common/example"
          Just s -> text $ T.decodeUtf8 s
      return ()
```

A função getConfig verifica e faz a leitura o conteúdo do arquivo. Um case é feito na expressão exampleConfig e Nothing representa a ausência deste arquivo, Just s representa que existe o arquivo e seu contéudo fica na variável s. A chamada de T.decodeUtf8 é simplesmente uma conversão entre duas representações diferentes de string.

Chapter 2

Reflex-dom

Vimos anteriormente as funções el e elAttr que são funções que criam elementos no Document Object Model da página. Pode-se criar alguns exemplos de elementos em funções fora da função frontend do capítulo anterior. Começaremos com alguns exemplos simples e depois falaremos mais sobre os tipos das funções envolvidas na criação de elementos no DOM de uma página

2.1 Exemplos

Alguns exemplos simples de interação em páginas e alguns eventos básicos serão listados para o melhor entendimento da framework.

Exemplo 2.1 *Uma lista de elementos como se fossem um menu.*

```
menu :: DomBuilder t m => m ()
menu = do
    el "div" £ do
        el "ul" £ do
            el "li" (text "Item 1")
            el "li" (text "Item 2")
            el "li" (text "Item 3")
            el "li" (text "Item 4")
```

Temos uma sucessiva aplicação das funções el, pois temos uma estruturas hierarquica de tags. A tag div seria a principal, a tag ul seria filha da div e as tags li filhas da tag ul. Os itens possuirão os textos indicados na função text. O html gerado por essa função fica próximo do trecho a seguir.

```
<div>
   <ul>
       <li> Item 1 </li>
       <li> Item 2 </li>
       <li> Item 3 </li>
```

```
        <li> Item 4 </li>
    </ul>
</div>
```

O tipo dfa função menu possui uma menção ao type class de vários parãmetros DomBuilder t m que indica uma restriçao na variável m que é uma monad atrelada a ele. Essa monad terá um retorno unitário (vazio, indicado pelo tipo ()). Essa função pode ser chamada em qualquer lugar da função frontend para vermos o efeito desejado. Caso quisermos marcar um dos itens com algum atributo, por exemplo, class usa-se o elAttr.

```
menu :: DomBuilder t m => m ()
menu = do
    el "div" £ do
        el "ul" £ do
            el "li" (text "Item 1")
            elAttr "li" ("class" =: "class1") (text "Item 2")
            el "li" (text "Item 3")
            el "li" (text "Item 4")
```

O trecho acima vai gerar a seguinte lista no html.

```
<div>
    <ul>
        <li> Item 1 </li>
        <li class="class1"> Item 2 </li>
        <li> Item 3 </li>
        <li> Item 4 </li>
    </ul>
</div>
```

Exemplo 2.2 *O segundo exemplo será um primeiro contato com um evento dinãmico. Ele consiste na escrita de duas palavras em duas caixas de texto e a mensagem digitada concatenada aparecerá do lado da caixa.*

```
caixas :: (DomBuilder t m, PostBuild t m) => m ()
caixas = el "div" £ do
  t <- inputElement def -- m (Dynamic Text)
  s <- inputElement def -- m (Dynamic Text)
  text " "
  dynText (zipDynWith (<>) (_inputElement_value t)
                           (_inputElement_value s) )
```

A função inputElement representa uma caixa de texto e a função _inputElement _value representa a extração desse valor dentro de um contexto de evento dinãmico do reflex (Dynamic Text). A função zipDynWith se comporta como um funtor aplicativo fazendo com que a função de dois parãmetros (<>) funcione em um conexto Dynamic nos possibilitando a concatenação das strings digitadas. A

função dynText representa a ação de um texto que é modificado em um curso de tempo através da digitação em alguma das duas caixas. Neste exemplo fica claro que está ocorrendo um eventListener do JavaScript por trás de tudo. Essa função também pode ser chamada em qualquer lugar da função frontend para vermos o efeito desejado.