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

Ужасно простой веб стек на Haskell

В Haskell есть большой выбор распространненных библиотек для всех простых нужд, от логирования доступа в базу данных до маршрутизации и подъема веб-сервера. Всегда хорошо иметь свободу выбора, но если вы просто начинаете, количество решений может сильно мешашть. Может быть вы даже еще не уверены, что вы способны понять важное различние между выборами. Вам нужно сделать запрос в бд. Вам нужны гарантирование строгие имена колонки и глубокое SQL встраивание которое Squeal дает вам, или вы возможно предпочтете относительную простоту или Opaleye с типобезопасностью? Или, может. лучше использовать postgresql-simple и оставить всё проще-простого? Или что насчет использования Selda? Или что на счет....

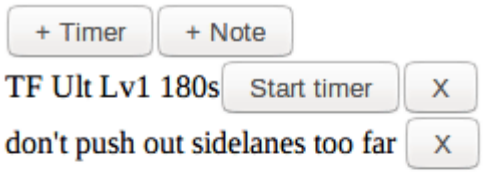
Возможность показать что вам не нужно проводить часы насколько ваш стек крут, - это возможность научиться самому. Я написал пример веб-приложения использующий самые простые библиотеки, которые я смог найти. Если вы не уверенны, как строить реальное приложение на Haskell, почему бы не начать с этого? Я умышленно постарался составить кодову базу максимально простой.

Пройдёмся по библиотекам которые я выбрал и что вы должны ожидать от них, ну и понять что это за приложене.

Хорошо, что же такое это ваше веб-приложеение?

Это сайт, где пользователи могут создать таймеры и записки.

Например, один из примеров использования может быть готовка: Кому-то нужно настройки различные таймеры для отслеживания прогресса различных реагентов, или составить заметки о вещах о которых необходимо беспокоиться, вещи которые могут быть улучшены в следующий раз повторя рецепт. Другой вариант может быть игра MOBA, как LoL или Dota2, где можно открыть страничку во втором мониторе для ослеживания кулдаунов, а так же записи о том как противостоять противникам и их кулдаунам во время битвы.



Давайте я покажу:

- Сессии, пользователи должны иметь возможность обновить страницу, или уйти и вернуться, а элементы должна всё еще оставаться.
- Постоянство и доступ к базе данных, нам нужно хранить таймеры и записки для каждого пользователя. Еще тонкость, это таймеры должны зранить оставшеся время. (Что если это 30 минутный таймер, а пользователь случайно закрыл вкладку?)
- Настройка во время работы, так как мы не можем хардкодить информацию о подключении к бд.
- Логирование. Само-собой разумеющееся для веб-приложения.

Исходный код приложения.

Что это за библиотеки?

Маршрутизация веб-сверера: Spock

Spock - из-за простоты использования. Если вы когда-либо пользовали Sinatra Ruby, Spock должен быть очень похож. Он так же идеть с обработкой сессии из коробки, что очень здорово.

Для примера, определим сервер с несколькими маршрутами для обратной отправки HTML и Json может выглядить следующим образом:

```
{-# LANGUAGE OverloadedStrings #-}

import Web.Spock as Spock
import Web.Spock.Config as Spock
import Data.Aeson as A

main :: IO ()
main = do
  spockCfg <- defaultSpockCfg () PCNoDatabase ()
  runSpock 3000 $ spock spockCfg $ do
    get root $ do
      Spock.html "<div>Hello world!</div>"
    get "users" $ do
      Spock.json (A.object [ "users" .= users ])
    get ("users" <//> var <//> "friends") $ \userID -> do
      Spock.json (A.object [ "userID" .= (userID :: Int), "friends" .= A.Null ])

  where users :: [String]
        users = ["bob", "alice"]
```

Доступ к базе данных: postgresql-simple

postgresql-simple - просто позволяет вам запустить SQL запрос к вашей базе данных, с минимумом дополнительных излишеств, таких как защита против injection-аттак. Он просто делает, что вам нужно, не больше.

```
{-# LANGUAGE OverloadedStrings #-}

import Database.PostgreSQL.Simple

userLoginsQuery :: Query
userLoginsQuery =
  "SELECT l.user_id, COUNT(1) FROM logins l GROUP BY l.user_id;"

getUserLogins :: Connection -> IO [(Int, Int)]
getUserLogins conn = query_ conn userLoginsQuery
```

Настройки: configurator

configurator читает конфигурационные файлы и парсит их в типы данных Haskell. Немного больше чем просто обычная читалка файлом конфига. имеет несколько трюков в рукаве. Конфигурационные атрибуты могут быть вложенными для группировки, так же конфигуратор предоставляет быструю перезагрузку при изменении настроек конфига, если это нужно.

Пример конфиг файла.

```
app_name = "The Whispering Fog"

db {
  pool {
    stripes = 4
    resource_ttl = 300
  }

  username = "pallas"
  password = "thefalloflatinium"
  dbname = "italy"
```

```
}
```

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Configurator as Cfg
import Database.PostgreSQL.Simple

data MyAppConfig = MyAppConfig
  { appName :: String
  , appDBConnection :: Connection
  }

getAppConfig :: IO MyAppConfig
getAppConfig = do
  cfgFile <- Cfg.load ["app-configuration.cfg"]
  name <- Cfg.require cfgFile "app_name"
  conn <- do
    username <- Cfg.require cfgFile "db.username"
    password <- Cfg.require cfgFile "db.password"
    dbname <- Cfg.require cfgFile "db.dbname"
    connect $ defaultConnectInfo
      { connectUser = username
      , connectPassword = password
      , connectDatabase = dbname
      }
  pure $ MyAppConfig
    { appName = name
    , appDBConnection = conn
    }
```

Логирование: fast-logger

fast-logger - предоставляет разумно простое в использовании средство логирования. В примере веб приложения я просто использую для вывода `stderr`, но у библиотеки есть возможность для логирования в файлы в том числе. Так как есть множество типов, в большинстве вы захотите определить функции-помошники которые просто принимают `LoggerSet` и сообщение которое нужно записать.

```
import System.Log.FastLogger as Log

logMsg :: Log.LoggerSet -> String -> IO ()
logMsg logSet msg =
  Log.pushLogStrLn logSet (Log.toLogStr msg)

doSomething :: IO ()
doSomething = do
  logSet <- Log.newStderrLoggerSet Log.defaultBufSize
  logMsg logSet "message 1"
  logMsg logSet "message 2"
```

Генерация HTML: blaze-html

Так как у нас не будет большого количества HTML, которое нужно будет генерировать в этом проекте, стоит упомянуть blaze-html for the parts that I did need.

Это естественно просто мелкое встраивание HTML в Haskell DSL. Если вы можете написать HTML, вы уже знаете как использовать библиотеку.

```
{-# LANGUAGE OverloadedStrings #-}

import Data.ByteString.Lazy

import Text.Blaze.Html5 as HTML
import Text.Blaze.Html5.Attributes as HTML hiding ( title )
import Text.Blaze.Html.Renderer.Utf8 as HTML

dashboardHTML :: HTML.Html
dashboardHTML = HTML.html $
  HTML.docTypeHtml $ do
    HTML.head $ do
      HTML.title "Timers and Notes"
      HTML.meta ! HTML.charset "utf-8"
      HTML.script ! HTML.src "/js/bundle.js" $ ""
    HTML.body $ do
      HTML.div ! HTML.id "content" $ ""
```

```
dashboardBytes :: ByteString
dashboardBytes = HTML.renderHtml dashboardHTML
```

Сборка и фротентд: make + npm

Да да, это не библиотеки. Но всё же, нам нужно что-то похожее на JavaScript фронтенд, так как таймеры должны обновляться в реальном времени. Webpack создает JS пакет, в то время как Make собирает результат приложения.

Я не хочу об этом много говорить. Есть множество источников про использование и того и другого инструмента.

Мне нужно это использовать?

Нет, конечно же нет. Если вы исследуете Haskell изначально, то вам возможно интересно. Не позволяйте мне вас удержать или диктовать что вы должны делать. Пока это приложение работает, многие части его могут считаться неидиоматическими для производства Haskell. Для примера, множество хаскеллят, скорей всего, будут использовать `Servant` вместо `Spock` для создания API точек доступа. Если вы заинтересовались еще чем-то, то должны следовать дальше.

Считайте эти библиотеки и это приложение как точку отсчёта. Я прошу вас использовать этот код как возможность изучить и понять как и что работает, затем начать мастерить. Одна из прекраснейших вещей про Haskell это то, насколько просто перерабатывать или обновлятся без проблем что-то сломать. Как только вы сделаете это приложение, почему бы не заменить части на более интересные библиотеки которые дают вам больше гарантий. как пусть постеменного изучения Haskell?

- Upgrade the DB access to use a type-safe query library instead of postgresql-simple. I recommend Opaleye!

- Upgrade the API definition to use Servant instead of Spock.
- Add automated testing using QuickCheck or hedgehog. For instance, you could test the property that every error response from the server also sends back a JSON error message. And you could even try replacing the frontend and build system.
- Upgrade the frontend code to use PureScript or Elm instead of vanilla JavaScript.
- Upgrade the build system to use Shake instead of Make to make things more robust.

Revision #5

Created 2022-01-21 13:01:38 UTC by gasick

Updated 2023-04-16 19:38:12 UTC by gasick