Beginning Web Programming in Haskell

Ollie Charles

May 29, 2015

Haskell is Ready for the Web!

If you want to do web programming Haskell, there has never been a better time!

- Many web frameworks: `snap`, `scotty`, `spock`, `happstack`, `yesod`, and others.
- DSLs for web languages: `blaze`, `lucid`, `jmacro`, `clay`
- Relational database abstractions: `opaleye`, `relational-record`, `persistant`
- Non-SQL databases: `hedis`, `acid-state`
- Serialization formats: `aeson`, `json-builder`, `yaml`

Haskell is Ready for the Web!

If you want to do web programming Haskell, there has never been a better time!

- Many web frameworks: `snap`, `scotty`, `spock`, `happstack`, `yesod`, and others.
- DSLs for web languages: `blaze`, `lucid`, `jmacro`, `clay`
- Relational database abstractions: `opaleye`, `relational-record`, `persistant`
- Non-SQL databases: `hedis`, `acid-state`
- Serialization formats: `aeson`, `json-builder`, `yaml`

It's also never been more daunting!

Today

- Today we'll build a small application allowing ZuriHac participants to share their projects they worked on this weekend
- Features:
  - People should be able to submit their projects with a basic description about the project
  - People can view a list of all ZuriHac projects and view details of each one
- Our stack will use...
  - `spock` as our underlying web framework
  - `lucid` to build HTML documents
  - `postgresql-simple` to interact with a PostgreSQL relational database
- I'll assume basic prior web programming experience, but welcome questions at any time

Setting up

I suggest using a cabal sandbox for this project:

```
mkdir zurihac-projects
cd zurihac-projects
cabal sandbox init
cabal update
cabal install spock postgresql-simple lucid
```

You can find all code for this online:

```
darcs get http://hub.darcs.net/ocharles/zurihac2015-projects-server
```

Spock in a Slide

Spock in a Slide

Spock in a Slide

Spock consists of:

- A basic HTTP server to serve applications
- A routing framework to define HTTP paths, with support for parameters and different HTTP verbs
- A domain specific language (DSL) for processing HTTP requests
  - Read information about the HTTP request - headers, request body, etc
  - Perform arbitrary IO such as interacting with databases
  - Deliver a HTTP response content and headers
- Built in support for sessions, application state and database connection pooling

Spock & monads

- You can do different things when you are routing or when you are delivering a request
- In Haskell, we can capture the differences in these effects by using different *monads*
- Spock has two main monads that capture these differences:
  1. In `SpockM` we can build a routing table for our application, and perform arbitrary IO
  2. In `SpockAction` we can inspect a HTTP request, perform arbitrary IO, and deliver HTTP responses.
- We define our application in the `SpockM` monad, and we use `SpockAction` by defining HTTP routes

Hello, Spock!

- Time for the "hello, world!" of web programming!

Hello, Spock!

- Time for the "hello, world!" of web programming!
- First, we define the logic to run for a web request, and specify what the response should be:

```
import Web.Spock.Simple

-- This SpockAction is parameterized to work with /any/ database,
-- session and application state.
helloSpock :: SpockAction database session state ()
helloSpock = do html "Hello, <em>Spock!</em>"
```

Hello, Spock!

- Time for the "hello, world!" of web programming!
- First, we define the logic to run for a web request, and specify what the response should be:

```
import Web.Spock.Simple

-- This SpockAction is parameterized to work with /any/ database,
-- session and application state.
helloSpock :: SpockAction database session state ()
helloSpock = do html "Hello, <em>Spock!</em>"
```

- Next, we *compose* our application to route `helloSpock` for GET requests:

```
app :: SpockM database session state ()
app = do get "/" helloSpock
```

Hello, Spock!

To run our application we have to specify:

- How sessions are used
- How to connect to our database
- What the application state is
- Which application to run, and the port to serve on

Hello, Spock!

To run our application we have to specify:

- How sessions are used
- How to connect to our database
- What the application state is
- Which application to run, and the port to serve on

For now, we'll just use dummy values, and revisit what these options mean later

Hello, Spock!

```haskell
main :: IO ()
main =
  do runSpock
       8000
       (spock sessionConfig dbConn initialState app)

sessionConfig :: SessionCfg ()
sessionConfig =
  SessionConfig "zurihac" (60 * 60) 0 True () Nothing

dbConn :: PoolOrConn ()
dbConn =
  PCConn (return ())
         (\_ -> return ())
         (PoolCfg 5 5 60)

initialState :: ()
initialState = ()
```

Rendering HTML with Lucid

- So far, we served HTML by writing HTML verbatim...

Rendering HTML with Lucid

- So far, we served HTML by writing HTML verbatim. . . yuck!

Rendering HTML with Lucid

- So far, we served HTML by writing HTML verbatim. . . yuck!
- Ideally, we want to work *in* Haskell as much as possible, avoiding strings
  - Syntax errors can be detected at compile time
  - We can encode what it means to be well-formed in types
  - We can use Haskell as a macro language and a means of abstraction
- There are many DSLs that do this, we'll look at `lucid`

Lucid

- In `lucid`, we can build a HTML tree using `do` notation
- `lucid` exports all HTML 5 elements, suffixed with an underscore
- To nest elements, we nest `do` blocks

```
import Lucid

helloSpockHTML :: HTML
helloSpockHTML =
  do html_
       (do head_ (do ...)
           body_ (do ...))
```

```
import Lucid

helloSpockHtml :: Html ()
helloSpockHtml =
  do html_
       (do head_ (do ...)
           body_ (do ...))
```

Lucid

- Text can be written as normal strings if we use the `OverloadedStrings` extension

```
{-# LANGUAGE OverloadedStrings #-}

 ...

helloSpockHtml :: Html ()
helloSpockHtml =
  do html_
       (do head_ (title_ "Hello!")
           body_
             (do h1_ "Hello!"
                 p_ "Hello, Lucid!"))
```

Lucid

- HTML attributes are lists
- Each attribute name is a function that expects a string argument

```
helloSpockHtml :: HTML
helloSpockHtml =
  do html_
       (do head_ (title_ "Hello!")
           body_
             (do h1_ "Hello!"
                 p_ "Hello, Lucid!"
                 p_ (do "I love"
                        a_ [href_ "http://haskell.org"]
                           "Haskell!")))
```

Lucid

Our HTML document is now just data, so we can refactor it

```
pageTemplate :: Html () -> Html ()
pageTemplate contents =
  do html_ (do head_ (title_ "Hello!")
               body_ contents)

link :: Text -> Html () -> Html ()
link url caption = a_ [href_ url] caption

helloSpockHtml :: Html ()
helloSpockHtml =
  pageTemplate
    (do h1_ "Hello!"
        p_ "Hello, Lucid!"
        p_ (do "I love "
               link "http://haskell.org" "Haskell!"))
```

Lucid & Spock

- `spock` only knows how to emit HTML text - it can't directly work with `lucid`
- In our application, we can *render* `lucid` as text

```
lucid :: Html () -> SpockAction database session state ()
lucid document = html (toStrict (renderText document))

helloSpock :: SpockAction database session state
helloSpock = do lucid helloSpockHTML
```

Connecting to a PostgreSQL database

- We've seen how to generate purely static pages, next we'll look at building dynamic pages based on the contents database.
- First, we need to change our application to connect to a database

```
import qualified Database.PostgreSQL.Simple as Pg

dbConn :: PoolOrConn Pg.Connection
dbConn =
  PCConn (ConnBuilder
            (Pg.connect
               Pg.defaultConnectInfo {Pg.connectUser = "zurihac"
                                     ,Pg.connectDatabase = "zurihac"})
            Pg.close
            (PoolCfg 5 5 60))
```

A data model

- In order to interact with the database, it's a good idea to have a domain-specific data model
- Our application will consist of projects. Each project has a name, a description, and a list of authors who worked on the project
- We can represent this in Haskell as a record:

```
data Project =
  Project {projectName :: Text
          ,projectDescription :: Text
          ,projectAuthors :: [Text]}
```

Writing Queries

- To write SQL queries, we can use the SQL *quasiquoter*
- This quasiquoter lets us write queries over multiple lines
- It also makes it purposely hard to concatenate queries, which means its harder to create SQL injection security holes.

```
{-# LANGUAGE QuasiQuotes #-}
import Database.PostgreSQL.Simple.SqlQQ (sql)

sqlListAllProjects :: Pg.Query
sqlListAllProjects =
  [sql| SELECT name, description, authors
        FROM projects
        ORDER BY name |]
```

- For a DSL, look into `opaleye`, `relational-record` or `HaskellDb`.

Running Queries

- In order to fetch data, we have to explain how to interpret each row
- `postgresql-simple` has a type class to specify how to marshal from PostgreSQL rows to Haskell data

```haskell
instance Pg.FromRow Project where
  fromRow = do
    name <- Pg.field
    description <- Pg.field
    authors <- fmap Vector.toList Pg.field
    return (Project name description authors)
```

Running Queries

To run queries, we can use the `query_` function

```
fetchAllProjects :: Pg.Connection -> IO [Project]
fetchAllProjects dbConn = Pg.query_ dbConn sqlListAllProjects
```

Running Queries in Spock

- To run a query, we needed a database connection.
- `spock` gives us the `runQuery` function, which will acquire a database connection from a connection pool.

```
--    This action needs a database connection
--                                        \/
getProjects :: SpockAction Pg.Connection session state ()
getProjects = do
  allProjects <- runQuery fetchAllProjects
  ...
```

Rendering Query Results as HTML

- Now that we have our query results as a Haskell data type, they are easy to render

```
projectToRow :: Project -> Html ()
projectToRow project =
  tr_ (do td_ (toHtml (projectName project))
          td_ (toHtml (projectDescription project))
          td_ (commaSeparate (map toHtml (projectAuthors project))))
  where
    commaSeparate :: [Html ()] -> Html ()
    commaSeparate = mconcat . intersperse ", "

renderProjects :: [Project] -> Html ()
renderProjects projects =
  table_ (do thead_ (tr_ (do th_ "Name"
                             th_ "Description"
                             th_ "Authors"))
             tbody_ (foldMap projectToRow projects))
```

A project listing action

```
getProjects :: SpockAction Pg.Connection session state ()
getProjects =
  do allProjects <- runQuery fetchAllProjects
     lucid (pageTemplate
              (do h1_ "Projects"
                  renderProjects allProjects
                  link "/add-project" "Add Your Project!"))
```

We also need to update app to route getProjects:

```
app :: SpockM Pg.Connection session state ()
app =
  do get "/" getProjects
```

POST parameters

- We need a way to allow users to submit their own projects to the database
- `spock` provides us with the `param` function to read `POST` data

```
param :: Text -> SpockAction database session state (Maybe Text)
```

- This function takes the name of a parameter and returns its value - if it can be found in the request
- Haskell is forcing us to be honest; this lookup could fail and we have to be ready for that!

Parse Submissions

We can now use `param` to pick out `POST` data:

```
projectFromPOST :: SpockAction database session state (Maybe Project)
projectFromPOST =
  do maybeName <- param "name"
     case maybeName of
       Just name ->
         do maybeDescription <- param "description"
            case maybeDescription of
              Just description ->
              ..
```

Parse Submissions

We can now use `param` to pick out `POST` data:

```
projectFromPOST :: SpockAction database session state (Maybe Project)
projectFromPOST =
  do maybeName <- param "name"
     case maybeName of
       Just name ->
         do maybeDescription <- param "description"
            case maybeDescription of
              Just description ->
              ..
```

What a mess! Can we abstract some of this out?

Accepting Submissions with MaybeT

- In Haskell we can chain a series of possibly-failing computations using the Maybe monad
- We can use a similar construction here with the MaybeT monad transformer
- We use MaybeT to introduce a step

```
MaybeT :: m (Maybe a) -> MaybeT m a

-- We leave 'MaybeT' by running it, which stops at the first failure
runMaybeT :: MaybeT m a -> m (Maybe a)
```

Accepting Submissions with MaybeT

```
projectFromPOST :: SpockAction database session state (Maybe Project)
projectFromPOST =
  runMaybeT
    (do name <-
          MaybeT (param "name")
        description <-
          MaybeT (param "description")
        authors <-
          sequence
            (map (\i -> MaybeT (param (pack ("author-" ++ show i))))
                 [0 .. 5])
        return (Project name description authors))
```

Adding projects to the database

To save projects to our database, we need a query to insert them:

```
sqlAddProject :: Query
sqlAddProject =
  [sql| INSERT INTO projects (name, description, authors)
        VALUES (?, ?, ?) |]
```

Add a way to run the query for a `Project`:

```
insertProject :: Project -> Pg.Connection -> IO ()
insertProject project dbConn =
  do Pg.execute dbConn sqlAddProject project
     return ()
```

A SpockAction to add =Project=s

We can now put all this together to build a SpockAction to add new projects

```
postProject :: SpockAction Pg.Connection session state ()
postProject =
  do maybeProject <- projectFromPOST
     case maybeProject of
       Nothing ->
         do lucid (p_ "Invalid submission")
            setStatus badRequest400
       Just project ->
         do runQuery (insertProject project)
            redirect "/"
```

Routing postProject

```
app :: SpockM Pg.Connection session state ()
app =
  do get "/" getProjects
     post "/projects" postProject
```

A Form to Add Projects

Finally, we provide a basic read-only action that gives the user a form to add a new project:

```
addProjectForm :: SpockAction database session state ()
addProjectForm =
  do lucid
       (pageTemplate
         (do form_
               [method_ "post",action_ "/projects"]
               (do p_ (do label_ "Project"
                          input_ [name_ "name"])
                   p_ (do label_ "Description"
                          input_ [name_ "description"])
                   mapM_ authorRow [0 .. 5]
                   input_ [type_ "submit" ,value_ "Add Project"])))
  where authorRow i =
          do p_ (do label_ (toHtml ("Author #" ++ show i))
                    input_ [name_ (pack ("author-" ++ show i))])
```

Wrapping Up

Let's recap what we've seen today:

- We can use `cabal` and `cabal sandbox` to download dependencies for our development environments.
- We saw how to use Spock to
  - Serve a HTTP application
  - Route the application in the `SpockM` monad
  - Respond to individual requests in the `SpockAction` monad
  - Inform Spock how to connect to our database and use the connection to run queries

Wrapping Up

- We used PostgreSQL to make our application dynamic:
  - A `data` model was used to represent the domain specific models
  - The `ToRow` and `FromRow` type classes to marshal this data to and from the database
  - The `sql` quasiquoter to embed SQL queries
  - Queries were executed using `query` and `execute`

Wrapping Up

- We assembled the UI for our application using Lucid, and learnt...
  - How to create HTML documents
  - How to add text with the `OverloadedStrings` extension
  - How Haskell can be used as a means of abstraction
- Finally, we learnt a little bit about good Haskell development with the `MaybeT` monad transformer

That's All, Folks!

That's everything I want to cover today, and hopefully you're now in a position where you're ready to start building some basic web applications :)