Fun With Haskell: ReST-ful Websites with Yesod

Nathaniel Wesley Filardo

January 18, 2012

Metadata
Questions?

- Questions from last time?

Metadata
Overview of today

- ReST-ful Web development with Yesod.
- First half: me going quickly through the Yesod Book [3] and QCon presentation [2]
- Second half: see if we can get something flying.

Introduction
Yesod?

Yesod ("foundation") is a full server-side web stack:

- Web server ("warp")
- Protocol for server/application connection ("WAI")
- Front-controller / router
- Template system ("Hamlet", "Lucius", "Julius")
- Database API ("yesod-persistent")
- With add-ons for more.

Introduction
Yesod?

Yesod aims to be:

- Fast!
- ReST-ful.
- Safe (using static typing and code generation)
  - Template system uses types to guard against XSS.
  - Type-safe URLs encode *every* URL on the site in Haskell data.
- Modular (this *is* Haskell).
- An *evolutionary* design: MVC, SQL DB integration, . . .

Introduction
ReST?

"Representational State Transfer." Defined by Roy Fielding in his 2000 thesis [1]. Roughly:

- Client-server architecture. Servers have resources that clients address.
- Servers store no per-client state.
- Explicit cache control on resources.
- Hidden server architecture ("am I talking to one or several servers?")

Introduction
ReST?

The upshot, as applied to HTTP:

- URIs uniquely name a resource (blog post, comment, user, ...)
- GET actions are read-only: return the latest description of the resource.
- DELETE and PUT are *idempotent* manipulators.
- POST more generally updates a resource.

Yesod gives us separate handlers for each (URI, Verb), rather than only routing on URI.

Introduction
Type-safe URLs?

- Every page on the site is identified by a piece of Haskell data.
    - That's really atypical: usually identified directly by path!
    - Algebraic data, in fact. Constructors take parameters!
- Rather than paste strings together, we use these handles and leave the rendering to Yesod.
- Dually, rather than tease strings apart, Yesod maps ("routes") URLs to data and hands them to us for *case analysis*!

Introduction
Hello World

- Let's look at the helloworld.hs example quickly.
- Gives us some idea of where we're going.
- Note: real Yesod sites use the "scaffolding" generated by `yesod init` which is much more feature-ful (multiple files!) and robust.

Introduction
Hello World

08-yesod/helloworld.hs

```
{-# LANGUAGE TypeFamilies,
             QuasiQuotes,
             MultiParamTypeClasses,
             TemplateHaskell,
             OverloadedStrings #-}
```

Alright, first off: we need a slew of extensions.

- Template Haskell and Quasi-Quotation are how Yesod will do its code generation on our behalf.
- Overloaded Strings let us use string literals in the same way as numeric literals. See `IsString` class.
- The others are type system extensions.

Introduction
Hello World

08-yesod/helloworld.hs

```
import Yesod
```

We need to import the modules we're going to use. For now, that's just Yesod itself.

Introduction
Hello World

08-yesod/helloworld.hs

```
data HelloWorld = HelloWorld

instance Yesod HelloWorld where
    approot _ = ""
```

We define a data type for our site (so simple, it doesn't take any parameters to construct one) and make this type an *instance* of the Yesod class. The `approot` class method is the root of the URI for our site; the empty string `""` works when we serve on the root of a site.

Introduction
Hello World

08-yesod/helloworld.hs

```
mkYesod "HelloWorld" [parseRoutes|
/ HomeR GET
|]
```

mkYesod is a Template Haskell function which blats out a lot of code for us. The funny `[parseRoute| |]` brackets are a **quasi-quoter** which generate code for us, too. We told it

- The map between URLs and Haskell data ("Home Resource")
- The methods which can be called on each

Introduction
Hello World

08-yesod/helloworld.hs

```
getHomeR :: Handler RepHtml
getHomeR = defaultLayout [whamlet|Hello World!|]
```

We have to say *what happens* when a GET request comes in for HomeR. We first use a "Hamlet widget" (we'll talk about those later) quasi-quoter (which actually makes a Builder) to capture the string; we then lay this out with `defaultLayout`, a method of the Yesod class.

Introduction
Hello World

08-yesod/helloworld.hs

```
main :: IO ()
main = warpDebug 3000 HelloWorld
```

Glue it all together. `warpDebug` is a really awesome utility function: given a port number and the data for a site, it sets up the warp web server and runs the site listening on localhost.

Shakespearian Templates

Yesod defines several “Shakespearian” template languages for generating web content. In order of increasing complexity:

- Julius for JavaScript.
- Cassius and Lucius for CSS.
- Hamlet for HTML.

All of these languages support **interpolation**, wherein we splice a Haskell value into the template.

Shakespearian Templates
Julius

- Julius in fact *only* supports interpolation.

Shakespearian Templates
Julius

- Julius in fact *only* supports interpolation.
- For example:

```
function(){#{f x} = "@{SomeR}";}
```

Shakespearian Templates
Julius

- Julius in fact *only* supports interpolation.
- For example:

```
function(){#{f x} = "@{SomeR}";}
```

- Shakespearian templates can reference a lot of things:

`#{x}`	The Haskell Text expression `x`.
`@{x}`	The URL path to the page computed by `x`.
`^{x}`	Splice in another template (of the same type) `x`.

Shakespearian Templates
Lucius

- Lucius is a strict superset of CSS.
- Supports interpolation and *nested* blocks.

```
article {
    code { background-color: grey; }
    p { text-indent: 2em; }
    a { text-decoration: none; }
}
```

Shakespearian Templates
Hamlet

Hamlet is a whitespace-based alternative to HTML. It supports interpolation:

```
<html>
  <head>
    <title>#{siteTitle} - Foo
    <link rel=stylesheet href=@{Stylesheet}>
  <body>
    <p>The subsequent material will amaze:
    ^{makeFancy}
```

Shakespearian Templates
Hamlet

It also supports some funky operators:

```
<body>
$with nvs <- null vs
  $if nvs
    $maybe alt <- mAlt
      <p>#{alt}
    $nothing
      <p>Sorry, nothing to display.
  $else
     <ul>
       $forall v <- vs
        <li>
          <a href=@{pageOf v}>#{v}
```

Shakespearian Templates
Hamlet

Hamlet also has lots of conveniences:

- Explicit whitespace markers if you need it.
- Convenience attributes for id (#) and class (.).
- DOCTYPE sugar.
- "simplified Hamlet" without support for URLs. ("shamlet")
- *Internationalized* Hamlet (w/ new interpolation)

See the documentation for more details, if you need them. (Also the `shakespeareTest.hs` file I pushed up.)

Widgets

Web programming requires that we manage three different languages: HTML, CSS, and JS.

- Great for single-page site: separate content, presentation rules, and client-side logic.
- In many-page sites, each page has to pick which CSS and JS their content requires.
- Makes providing reusable chunks of “a website” difficult.

Widgets

Yesod provides a `Widget` for encapsulating content and its required “stuff”. A widget describes

- The title
- CSS (external references and internal declarations)
- JS (ditto)
- Other tags in the `<head>`
- Other tags in the `<body>`

Widgets

Some primitive widget combinators (non-exhaustive list):

- `setTitle` takes a chunk of Html and makes it the title.
- Adding scripts: `addScript` (type-safe URL), `addScriptRemote` (arbitrary URL)
- `toWidget` is overloaded on type; Hamlet goes in the body, Julius inside `<script>`, Lucius in `<style>`.

Widgets

And! (Wait for it...)

Widgets

And! (Wait for it...)

- Widget is a Monad.

Widgets

And! (Wait for it. . .)

- Widget is a Monad.
- And a Monoid.

Widgets

And! (Wait for it. . .)

- Widget is a Monad.
- And a Monoid.
- Combine widgets into an überwidget using do notation!

```
uwidget = do
  setTitle "If you didn’t set one before..."
  toWidget [hamlet|<h1>Really Big Heading|]
  toWidget [hamlet|<h5>Sub-heading under that|]
  toWidget [lucius|h1 { color : green } |]
```

Widgets

Other widget niceties:

- `newIdent` operator for making a unique name, say for class labels.
- The `whamlet` quasi-quoter: like `hamlet` except that
  - It produces a widget.
  - the embedding interpolation (`^{...}`) now also takes *widgets*.

Widgets

- In fact, essentially everything gets turned into a Widget either explicitly or internally.
- Then, the `defaultLayout` method of the `Yesod` class is given the whole widget hierarchy and renders it.
- We didn't specify one in `helloworld.hs` (so we used the default) but we can override it. This is how Yesod site-theming works.
- See the documentation for details.

The Yesod Type Class

On that note, there's a lot to say about the Yesod class itself.

- Path handling
- Default layout
- Error pages
- Automatic handling of static CSS and JS
- Messages
- Authentication

All I will say is this: there's documentation if you want it, and you probably will, but maybe not in the next two hours.

Routing

- Previously alluded to having multiple pages. How do we actually do that?
- Recall:

08-yesod/helloworld.hs

```
mkYesod "HelloWorld" [parseRoutes|
/ HomeR GET
|]
```

- The quoted line says “The path / corresponds to the `HomeR` resource and supports the GET method”
- Elsewhere, we defined `getHomeR`.

Routing

- **Static paths** we've seen:

```
/     HomeR GET
/a/b SomeR
```

- **Dynamic single paths** take a type denoted with #:

```
/def/#String   DefR GET POST DELETE
/sum/#Int/#Int SumR GET
```

- **Dynamic multi paths** take a *:

```
/wiki/*Texts   WikiR GET
```

- Also **subsites**: see documentation.

Routing

Remember, the quasi-quoter is building an ADT for our site.

- Static paths are constructors with no arguments.
- Dynamic paths are constructors with an argument for each match.
- Type classes for (de)coding: `SinglePiece` and `MultiPiece` if you want to define your own match types.

That's not to say that every piece of data of this type is a page; it's just a *valid URL*.

Routing

Yesod handles the details of matching and so on automatically. Then what does it do?

- If you leave off the list of methods (e.g. `/ FooR`), you get a single callback for all URLs that matched, called `handleFooR`.
- If you give the list of methods (e.g. `/ FooR GET POST`), you get discriminated callbacks: `getFooR`, `postFooR`. Anything you didn't mention gets 405 treatment.
- Again, also subsites. See documentation.

Routing

- The matches in the routing declarations are arguments to the handler functions.
  - Static handlers don't take any
  - Dynamic handlers take one per match of the right type.
- Handlers' return type is `HasReps a => Handler a`: a `Handler`-monadic action returning some response.
- Typical responses are `RepHtml`, `RepPlain`, `RepJson`.

Routing

Handler has lots of things you might want.

- Access to information about the site (getYesod)
- Access to request information (lookupGetParam, lookupCookie, getRequest).
- Response header control: setCookie, cacheSeconds, ...
- Short-circuiting behavior for
  - redirect to a type-safe URL
  - notFound and other errors
  - sendFile for static files
- Again, documentation is great.

Client-side Session State

- Try as we might, sometimes we just can't ReST.
    - Typical examples: login, shopping carts.
- Sometimes, we want a per-client key/value store.
    - Ideally, not loading our database.
- Use HTTP cookies.
    - With encryption and MACing for security!
    - Handled by the `clientsession` package.

Client-side Session State

Really simple, `Handler`-monadic API:

- Set a session key's value with `setSession k v`.
- Get with `lookupSession k`, which returns a `Maybe`.
- Delete with `deleteSession k`.

(Types elided for simplicity.)

Client-side Session State
Messages

Sometimes we want to tell the user something on the *next* page load (e.g. after handling a POST request and redirecting the user). **Messages** give us a way to do this easily:

- `setMessage` to make the note to ourselves.
- `getMessage` to get the message and clear it.
- Suggested that getMessage happen in `defaultLayout` so that it "just happens" by default.

Server-side Persistent State

Ah, the moment you've all been waiting for.

- Hooking Haskell up to a database.
- Details handled for us by the `persistent` package.
  - Non-relational, database-agnostic system.
  - For today: sqlite backend.
  - Also PostgreSQL and MongoDB and room for more.
  - Capable of handling (some) migrations automatically.
- I am giving you the most basic stuff.

Server-side Persistent State

Here's what we need to do:

- Define our database schema using a quasi-quoter or two.
- Define and use a pool of database connections.
- Run database commands in handlers.

I'm going to use the example from the end of the book chapter on persistent, which is also `PersistTest.hs` on the course website.

Server-side Persistent State
Defining the Schema

08-yesod/PersistTest.hs

```
share [mkPersist sqlSettings,
      mkMigrate "migrateAll"] [persist|
Person
    firstName String
    lastName String
    age Int Gt Desc
|]
```

Defines `PersonFirstName`, `PersonLastName`, and `PersonAge` columns and types. Further, implicitly defines a `PersonId` column and type.

Server-side Persistent State
Pool Management

First things first, our foundation needs to carry the database pool around:

08-yesod/PersistTest.hs

```
data PersistTest = PersistTest ConnectionPool
```

Contrast to

08-yesod/helloworld.hs

```
data HelloWorld = HelloWorld
```

Server-side Persistent State
Pool Management

Need to tell Yesod a few things. We make our foundation type an instance of `YesodPersist`:

```
instance YesodPersist PersistTest where
```

- Need to pick a particular backend (using "associated types"; cool stuff!)

```
    type YesodPersistBackend PersistTest
                              = SqlPersist
```

Server-side Persistent State
Pool Management

```
instance YesodPersist PersistTest where
```

- Also need to define how to run DB operations:

```
    runDB action = liftIOHandler $ do
        PersistTest pool <- getYesod
        runSqlPool action pool
```

  - “Get the foundation and pattern match out the pool”
  - “Run our action against that pool”
  - “Lift into the right monad with `liftIOHandler`”

Server-side Persistent State
Pool Management

When we start up, build a connection pool, run migrations, and then give the pool to our foundation:

```
main = withSqlitePool "test.db3" 10 $ \pool -> do
    runSqlPool (runMigration migrateAll) pool
    {- ... -}
    warpDebug 3000 $ PersistTest pool
```

Server-side Persistent State
Running Database Operations

Actually running database operations is now easy:

```
getPersonR :: PersonId -> Handler RepPlain
getPersonR personId = do
    person <- runDB $ get404 personId
    return $ RepPlain $ toContent $ show person
```

We make use of the `runDB` we just defined and the `get404` utility function (either gives us the requested object, or short-circuits with a 404). Code knows the right column to use from the type annotation.

Form Handling

- Hoo boy, forms are big. Probably too much for now.
- Manage all sorts of things in a nice API:
    - Server-side validation
    - Marshalling to/from strings ("boundary problem")
    - Generate HTML and JavaScript fun stuff.
    - JS client-side validation (just for UX)
    - Automatic form-field name generation.
    - Anti-CSRF nonces
- As usual, forms are compositional.

Scaffolding

- Real websites don't fit entirely in one file.
- Run `yesod init` to get a skeleton multi-file website.
- cd into the directory it made, run `yesod devel`.
- Then visit `http://localhost:3000/`.
- And look around, at `Foundation.hs` first.

Next time

- Alright, with that done. . . all yours.
- People should mail me with suggestions for Friday:
    - More me talking about Yesod?
    - More you working on your stuff?
    - Lambda calculus and category theory?

Bib

- Roy Thomas Fielding.
  Architectural Styles and the Design of Network-based Software Architectures.
  PhD thesis, 2000.
  Available from: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

- Michael Snoyman.
  Yesod web framework, November 2011.
  QCon, San Francisco.
  Available from: https://docs.google.com/present/view?id=dz4jvnj_54hrjnwpdc.

- Michael Snoyman.
  Yesod Web Framework Book.

2012.
Available from: http://www.yesodweb.com/book.