The Happstack Book: Modern, Type-Safe Web Development in Haskell

Jeremy Shaw

Contents

Introduction

Happstack is a family of libraries for creating fast, modern, and scalable web applications. A majority of the libraries are loosely coupled so that you can choose the technologies that are best for your particular application. You are not required to use any particular library for your database, templating, routing, etc.

However, if you are new to Haskell web development, too many choices can be overwhelming. So, there are three places you might consider starting.

`happstack-lite` is the quickest and easiest way to get started. It uses `blaze-html` for HTML generation and simple dynamic routing combinators. It has a very small, but sufficient API, for writing many web applications.

`clckwrks` is a higher-level web framework built on top of the same technology that `happstack-foundation` uses. `clckwrks` makes it easy to develop web applications by providing off-the-shelf plugins and themes. It even has (very) experimental support for installing new themes and plugins with out restarting the server. clckwrks plugins can do just about anything. Current plugins include a CMS/blogging system, a media gallery, an ircbot, and more.

This book covers the many libraries commonly used by Happstack developers. You can feel free to skip around and read just the sections you are interested in. Each section comes with a small, self-contained example that you can download and run locally.

This book was recently converted to a new build system. If you find any mistakes, formatting errors, bad links, etc, please report them here, https://github.com/Happstack/happstack-book/issues.

In addition to the HTML version of this book you can also read it in PDF and EPUB/NOOK.

Hello World

Your first app!

Our first Happstack application is a simple server that responds to all requests with the string, `Hello, World!`.

```
module Main where

import Happstack.Server (nullConf, simpleHTTP, toResponse, ok)

main :: IO ()
main = simpleHTTP nullConf $ ok "Hello, World!"
```

If you want to run the code locally, and you have not already installed Happstack – you will need to do that first. You can find instructions on how to install Happstack at http://happstack.com/page/view-page-slug/2/download.

To build the application run:

```
$ ghc --make -threaded HelloWorld.hs -o helloworld
```

The executable will be named `helloworld`. You can run it like:

```
$ ./helloworld
```

Alternatively, you can use `runhaskell` and avoid the compilation step:

```
$ runhaskell HelloWorld.hs
```

Run this app and point your browser at `http://localhost:8000/`. (assuming you are building the program on your local machine.) The page should load and say `"Hello, World!"`.

Alternatively, we can use `curl`:

```
 $ curl http://localhost:8000/
Hello, World!
```

`curl` is a command-line utility which can be used to create many types of HTTP requests. Unlike a browser, it does not attempt to render the results, it just

prints the body of the response to the console.

If you run `curl` with the `-v` option it will provide verbose output which includes the headers sent back and forth between curl and the server:

```
$ curl -v http://localhost:8000/
* About to connect() to localhost port 8000 (#0)
*   Trying 127.0.0.1... connected
> GET / HTTP/1.1
> User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu)
> Host: localhost:8000
> Accept: */*
>
< HTTP/1.1 200 OK
< Transfer-Encoding: chunked
< Connection: Keep-Alive
< Content-Type: text/plain; charset=UTF-8
< Date: Thu, 13 Dec 2012 00:19:01 GMT
< Server: Happstack/7.0.7
<
* Connection #0 to host localhost left intact
* Closing connection #0
Hello, World!
```

This can sometimes be useful for debugging why your site is not working as you expect.

`curl` is not required by Happstack or this book, but it is a useful tool for web development. `curl` is not part of Happstack. The official curl website is http://curl.haxx.se.

The parts of `Hello World`

Listening for HTTP requests

The `simpleHTTP` function is what actually starts the program listening for incoming HTTP requests:

```
simpleHTTP :: (ToMessage a) => Conf -> ServerPartT IO a -> IO ()
```

We'll examine the various parts of this type signature in the following sections.

Configuring the HTTP listener

The first argument, `Conf`, is some simple server configuration information. It is defined as:

```
data Conf = Conf
    { port       :: Int
    , validator  :: Maybe (Response -> IO Response)
```

```
    , logAccess  :: forall t. FormatTime t => Maybe (LogAccess t)
    , timeout    :: Int
    }
```

The fields can be described as:

port the TCP port to listen on for incoming connection

validator on-the-fly validation of output during development

logAccess logging function for HTTP requests

timeout number of seconds to wait before killing an inactive connection

The default config is `nullConf` which is simply defined as:

```
-- | Default configuration contains no validator and the port is set to 8000
nullConf :: Conf
nullConf = Conf
    { port      = 8000
    , validator = Nothing
    , logAccess = Just logMAccess
    , timeout   = 30
    }
```

Processing a `Request`

If we imagined a stripped-down web server, the user would just pass in a handle function with the type:

```
Request -> IO Response
```

where `Request` and `Response` correspond to HTTP requests and response. For every incoming request, the server would fork off a new thread thread and call the handler.

While this would work – the poor developer would have to invent all manner of adhoc mechanisms for mapping routes, adding extra headers, sending compressed results, returning errors, and other common tasks.

Instead ,`simpleHTTP` takes a handler with the type:

```
(ToMessage a) => ServerPartT IO a
```

There are three key differences:

1. The `ServerPartT` monad transformer adds a bunch of functionality on top of the base IO monad
2. the `Request` argument is now part of `ServerPartT` and can be read using `askRq`.
3. The `ToMessage` class is used to convert the return value to a `Response`.

`simpleHTTP` processes each incoming request in its own thread. It will parse the `Request`, call your `ServerPartT` handler, and then return a `Response` to the client. When developing your handler, it is natural to think about things as if you are writing a program which processes a single `Request`, generates a single `Response`, and exits. However it is important when doing I/O, such as writing files to disk, or talking to a database to remember that there may be other threads running simultaneously.

Setting the HTTP response code

In this example, our handler is simply:

```
ok "Hello, World!" :: ServerPartT IO String
```

`ok` is one of several combinators which can be used to set the HTTP response code. In this case, it will set the response code to `200 OK`. The type signature for `ok` can be simplified to:

```
ok :: a -> ServerPartT IO a
```

`ok` acts much like `return` except it also sets the HTTP response code for a `Response`.

`Happstack.Server.SimpleHTTP` contains similar functions for the common HTTP response codes including, `notFound`, `seeOther`, `badRequest` and more.

Creating a `Response`

The `ToMessage` class is used to turn values of different types into HTTP responses. It contains three methods:

```
class ToMessage a where
  toContentType :: a -> ByteString
  toMessage     :: a -> Lazy.ByteString
  toResponse    :: a -> Response
```

A vast majority of the time we only call the `toResponse` method.

`simpleHTTP` automatically calls `toResponse` to convert the value returned by the handler into a `Response` – so we did not have to call `toResponse` explicitly. It converted the `String "Hello, World!"` into a `Response` with the content-type `"text/plain"` and the message body `"Hello, World!"`

Often times we will opt to explicitly call `toResponse`. For example:

```
-- / show
module Main where

import Happstack.Server (nullConf, simpleHTTP, toResponse, ok)
-- show
main :: IO ()
main = simpleHTTP nullConf $ ok (toResponse "Hello, World!")
```

Happstack comes with pre-defined `ToMessage` instances for many types such as `Text.Html.Html`, `Text.XHtml.Html`, `String`, the types from HSP, and more.

Source code for this app is here.

Choosing between multiple `ServerPartTs`

In the first example, we had only one `ServerPartT`. All `Requests` were handled by the same part and returned the same `Response`.

In general, our applications will have many `ServerPartTs`. We combine them into a single `ServerPartT` by using `MonadPlus`. Typically via the `msum` function:

```
msum :: (MonadPlus m) => [m a] -> m a
```

In the following example we combine three `ServerPartTs` together.

```
module Main where

import Control.Monad
import Happstack.Server (nullConf, simpleHTTP, ok, dir)

main :: IO ()
main = simpleHTTP nullConf $ msum [ mzero
                                  , ok "Hello, World!"
                                  , ok "Unreachable ServerPartT"
                                  ]
```

The behaviour of `MonadPlus` is to try each `ServerPartT` in succession, until one succeeds.

In the example above, the first part is `mzero`, so it will always fail. The second part will always succeed. That means the third part will never be reachable.

Alas, that means this application will appear to behave exactly like the first application. What we need are some ways to have parts match or fail depending on the contents of the HTTP `Request`.

Source code for this app is here.

Route Filters

a.k.a Responding to different url paths

Happstack provides a variety of ways to match on parts of the `Request` (such as the path or request method) and respond appropriately.

Happstack provides two different systems for mapping the request path to a handler. In this section we will cover a simple, untyped routing system. Later we will look at fancier, type-safe routing sytem known as `web-routes`.

Using `dir` to match on static path components

We can use `dir` to handle components of the URI path which are static. For example, we might have a site with the two URLs: `hello` and `goodbye`.

```
module Main where

import Control.Monad
import Happstack.Server (nullConf, simpleHTTP, ok, dir, seeOther)

main :: IO ()
main = simpleHTTP nullConf $ msum
    [ dir "hello"    $ ok "Hello, World!"
    , dir "goodbye"  $ ok "Goodbye, World!"
    , seeOther "/hello" "/hello"
    ]
```

If we start the app and point our browser at http://localhost:8000/hello we get the `hello` message, and if we point it at http://localhost:8000/goodbye, we get the `goodbye` message.

Source code for this app is here.

Using `dir` to match on multiple components

We can match on multiple components by chaining calls to `dir` together:

```
module Main where

import Control.Monad    (msum)
import Happstack.Server (dir, nullConf, ok, seeOther, simpleHTTP)

main :: IO ()
main = simpleHTTP nullConf $
  msum [ dir "hello"   $ dir "world" $ ok "Hello, World!"
       , dir "goodbye" $ dir "moon"  $ ok "Goodbye, Moon!"
       , seeOther "/hello/world" "/hello/world"
       ]
```

If we start the app and point our browser at http://localhost:8000/hello/world we get the hello message, and if we point it at http://localhost:8000/goodbye/moon, we get the goodbye message.

Source code for this app is here.

Using `dirs` as shorthand to match on multiple components

As a shorthand, we can also use `dirs` to handle multiple static path components.

```
module Main where

import Control.Monad    (msum)
import Happstack.Server (dirs, nullConf, ok, seeOther, simpleHTTP)

main :: IO ()
main = simpleHTTP nullConf $
  msum [ dirs "hello/world"  $ ok "Hello, World!"
       , dirs "goodbye/moon" $ ok "Goodbye, Moon!"
       , seeOther "/hello/world" "/hello/world"
       ]
```

If we start the app and point our browser at http://localhost:8000/hello/world we get the hello message, and if we point it at http://localhost:8000/goodbye/moon, we get the goodbye message.

Source code for this app is here.

Matching on variable path segments

Often times a path segment will contain a variable value we want to extract and use, such as a number or a string. We can use the `path` combinator to do that.

```
path :: (FromReqURI a, MonadPlus m, ServerMonad m) =>
        (a -> m b) -> m b
```

You may find that type to be a little hard to follow because it is pretty abstract looking. Fortunately, we can look at it in an easier way. A `ServerPart` is a valid instance of, `ServerMonad m`, so we can just replace the `m` with `ServerPart`. You can do this anywhere you see type signatures with `(ServerMonad m) =>` in them. In this case, the final result would look like:

```
path :: (FromReqURI a) => (a -> ServerPart b) -> ServerPart b
```

`path` will attempt to extract and decode a path segment, and if it succeeds, it will pass the decoded value to the nested server part.

Let's start with the most basic example, extracting a `String` value. We will extend the Hello World server so that we can say hello to anyone.

```
module Main where

import Control.Monad    (msum)
import Happstack.Server (nullConf, simpleHTTP, ok, dir, path, seeOther)

main :: IO ()
main = simpleHTTP nullConf $
  msum [ dir "hello" $ path $ \s -> ok $ "Hello, " ++ s
       , seeOther "/hello/Haskell" "/hello/Haskell"
       ]
```

Now, if we start the app and point our browser at: http://localhost:8000/hello/World we get the response `"Hello, World"`. if we point it at http://localhost:8000/hello/Haskell, we get `"Hello, Haskell"`.

Source code for this app is here.

FromReqURI: extending path

We can extend `path` so that we can extract our own types from the path components as well. We simply add an instance to the `FromReqURI` class:

```
class FromReqURI a where
    fromReqURI :: String -> Maybe a
```

Let's look at an example:

```
module Main where

import Control.Monad    (msum)
import Data.Char        (toLower)
import Happstack.Server ( FromReqURI(..), nullConf, simpleHTTP
```

```
                        , ok, dir, path, seeOther
                        )
```

let's say that we want to create a type to represent subjects we can greet:

```
data Subject = World | Haskell

sayHello :: Subject -> String
sayHello World   = "Hello, World!"
sayHello Haskell = "Greetings, Haskell!"
```

Then we simply add a `FromReqURI` instance:

```
instance FromReqURI Subject where
    fromReqURI sub =
        case map toLower sub of
          "haskell" -> Just Haskell
          "world"   -> Just World
          _         -> Nothing
```

Now when we use `path` it will extract a value of type `Subject`.

```
main :: IO ()
main = simpleHTTP nullConf $
  msum [ dir "hello" $ path $ \subject -> ok $ (sayHello subject)
       , seeOther "/hello/World" "/hello/World"
       ]
```

Now, if we start the app and point our browser at: http://localhost:8000/hello/World we get the response `"Hello, World"`. if we point it at http://localhost:8000/hello/Haskell, we get `"Greetings, Haskell!"`.

Source code for this app is here.

Matching on request method (`GET`, `POST`, `etc`)

We can specify that a route is only valid for specific HTTP request methods by using the `method` guard:

```
method :: (ServerMonad m, MonadPlus m, MatchMethod method) => method -> m ()
```

Here is a simple demo app:

```
module Main where

import Control.Monad    (msum)
import Happstack.Server ( Method(GET, POST), dir, method
                        , nullConf, ok, simpleHTTP, seeOther
```

```
                        )

main :: IO ()
main = simpleHTTP nullConf $ msum
       [ do dir "foo" $ do method GET
                           ok $ "You did a GET request on /foo\n"
       , do method GET
            ok $ "You did a GET request.\n"
       , do method POST
            ok $ "You did a POST request.\n"
       ]
```

Using `curl` we can see the expected results for normal `GET` and `POST` requests to `/`:

```
 $ curl http://localhost:8000/
You did a GET request.
 $ curl -d '' http://localhost:8000/
You did a POST request.
```

Note that `method` does not require that all the segments of request path have been consumed. We can see in here that `/foo` is accepted, and so is `/foo/bar`.

```
 $ curl http://localhost:8000/foo
You did a GET request on /foo
 $ curl http://localhost:8000/foo/bar
You did a GET request on /foo
```

You can use `nullDir` to assert that all the path segments have been consumed:

```
nullDir :: (ServerMonad m, MonadPlus m) => m ()
```

Source code for this app is here.

Advanced method matching with `MatchMethod`

The method routing functions use a class `(MatchMethod method)` instead of the concrete type `Method`. The `MatchMethod` class looks like this:

```
class MatchMethod m where
    matchMethod :: m -> Method -> Bool

instance MatchMethod Method           where ...
instance MatchMethod [Method]         where ...
instance MatchMethod (Method -> Bool) where ...
instance MatchMethod ()               where ...
```

This allows us to easily match on more than one method by either providing a list of acceptable matches, or by providing a function which returns a boolean

value. We can use this feature to support the `HEAD` method. When the client does a `HEAD` request, the server is supposed to return the same headers it would for a GET request, but with an empty response body. Happstack includes special support for handling this automatically in most cases.

```
module Main where

import Control.Monad (msum)
import Happstack.Server ( Method(GET, HEAD), dir, methodM
                        , nullConf, ok, simpleHTTP
                        )

main :: IO ()
main = simpleHTTP nullConf $ msum
       [ do methodM [GET, HEAD]
            ok $ "Hello, World\n"
       ]
```

We can now use curl to do a normal `GET` request, or we can use the `-I` flag which does a `HEAD` request:

```
 $ curl http://localhost:8000/
Hello, World
 $ curl -I http://localhost:8000/
HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Length: 13
Content-Type: text/plain; charset=UTF-8
Date: Tue, 15 Jun 2010 19:56:07 GMT
Server: Happstack/0.5.0
```

Happstack automatically notices that it is a `HEAD` request, and does not send the body.

Source code for this app is here.

Other Routing Filters

SimpleHTTP includes a number of other useful routing filters, such as:

`nullDir :: (ServerMonad m, MonadPlus m) => m ()` check that there are no path segments remaining `host :: (ServerMonad m, MonadPlus m) => String -> m a -> m a`
match on a specific host name in the Request `withHost :: (ServerMonad m, MonadPlus m) => (String -> m a) -> m a`
Lookup the host header and pass it to the handler. `uriRest :: (ServerMonad m) => (String -> m a) -> m a`

Grab the rest of the URL (dirs + query) and passes it to your handler
`anyPath :: (ServerMonad m, MonadPlus m) => m r -> m r`
match on any path ignoring its value `trailingSlash :: (ServerMonad m, MonadPlus m) => m ()`
Guard which checks that the Request URI ends in `/`. Useful for distinguishing between `foo` and `foo/`

Templating for HTML and Javascript

Happstack supports a number of third party templating and HTML libraries. It is easy to add support for additional libraries, if your favorite does not already have support.

The three most popular choices are `HSP`, `blaze-html`, and `heist`.

`blaze-html` is a fast, combinator based HTML creation library.

pros:

- Claims to be fast (some benchmarks to back this up)
- Use of combinators ensures output is always well-formed and free of typos in the names of elements and attributes
- Automatic escaping of String values
- Able to use the power of Haskell in your templates
- Type-checked at compile time to ensure no template values are missing
- Nice syntax (compared to the old html and xhtml libraries.)

cons:

- Requires you to recompile in order to update the template
- Makes it easy to mix the application logic and view code together, making it hard to update later (therefore you must have self control)
- Only suitable for generating HTML documents
- Not ideal for having templates written by a web designer who does not know Haskell
- No compile-time assurances that generated html/xml is valid (though it will be well-formed).
- The `Html` monad is not a real monad, nor is it a monad transformer. This eliminates some advantage usage possibilities.

`HSP` allows you to embed literal XML syntax inside your Haskell code. A pre-processor or `QuasiQuoter` rewrites the literal XML into normal haskell function calls, and then the code is compiled.

pros:

- Templates are compiled, so they are pretty fast (needs more benchmarks to support this statement however)
- You have the full power of Haskell in your templates, because it is Haskell (with a purely syntactic extension)
- Type-checked at compile time to ensure types are correct and no template values are missing
- Automatic escaping of String values
- Syntax is very similar to XML/HTML, so it is easy to learn
- Can be easier to work with when trying to populate a template from a complex Haskell type
- Can be used to generate HTML or XML

cons:

- Requires you to recompile in order to update the template
- Error messages are sometimes misleading or hard to understand
- Makes it easy to mix the application logic and view code together, making it hard to update later (therefore you must have self control)
- Only suitable for generating XML and HTML documents
- Not ideal for having templates written by a web designer who does not know Haskell (although the xml syntax helps)
- No compile-time assurances that generated html/xml is valid (though it will be well-formed).

`Heist` uses a combination of external XML files and Haskell code to perform templating.

pros:

- changes to the external XML files can be reloaded with out having to recompile and restart the server
- a large portion of the template system is standard XHTML in external templates, making it easier for web designers to use

cons:

- prone to silent runtime errors

Using `blaze-html`

It is trivial to use `blaze-html` with Happstack. Essentially you just use `toResponse` to convert a blaze `Html` value into a `Response`. For more detailed information on using `blaze-html`, see the `blaze-html website`. The following example should get you started:

```
{-# LANGUAGE OverloadedStrings #-}
module Main where
```

```
import Happstack.Server
import           Text.Blaze ((!))
import qualified Text.Blaze.Html4.Strict as H
import qualified Text.Blaze.Html4.Strict.Attributes as A

appTemplate :: String -> [H.Html] -> H.Html -> H.Html
appTemplate title headers body =
    H.html $ do
      H.head $ do
        H.title (H.toHtml title)
        H.meta ! A.httpEquiv "Content-Type"
               ! A.content "text/html;charset=utf-8"
        sequence_ headers
      H.body $ do
        body

helloBlaze :: ServerPart Response
helloBlaze =
   ok $ toResponse $
    appTemplate "Hello, Blaze!"
                [H.meta ! A.name "keywords"
                        ! A.content "happstack, blaze, html"
                ]
                (H.p $ do "Hello, "
                          H.b "blaze-html!")

main :: IO ()
main = simpleHTTP nullConf $ helloBlaze
```

Source code for the app is here.

Now if we visit http://localhost:8000/, we will get an HTML page which says:

Hello, blaze-html!

This example is pretty simple, but there are a few things to note:

- The `appTemplate` function is purely `blaze-html` code and is in no way Happstack specific.
- The existence of the `appTemplate` is purely a stylistic choice.
- I have found it useful to set the content-type meta tag.
- Happstack will automatically set the HTTP header `Content-Type: text/html; charset=UTF-8`. (`blaze-html` only supports UTF-8)

Using HSX/HSP

To enable HSX support, you must install the `happstack-hsp` package.

HSX is an XML-based templating system that allows you to embed XML in your Haskell source files. If you have ever had to use PHP, you may want to run screaming from this idea. However, the HSX solution is far saner than the PHP solution, so you may want to give it a chance.

There are two ways you can use `hsx`. One way is to use an external preprocessor `hsx2hs`. The other way is to use the `[hsx| |]` quasiquoter.

hsx2hs

The `hsx2hs` is the traditional way of embedding literal XML into Haskell. It predates the existance of the `QuasiQuotes` extension.

There are two benefits to the `hsx2hs` preprocessor over the `QuasiQuotes`:

1. it does not require extra syntax to delimit where the XML starts
2. it can be used with Haskell compilers that do not support `QuasiQuotes` such as `Fay`.

However it has a few drawbacks as well:

1. it strips haddock comments from the source file
2. it can screw up the line numbers in error messages

Those drawbacks are fixable, but require some serious effort.

The first thing we will see is a funny `OPTIONS_GHC` pragma at the top of our file:

```
{-# LANGUAGE FlexibleContexts, OverlappingInstances #-}
{-# OPTIONS_GHC -F -pgmFhsx2hs #-}
module Main where
```

This pragma at the top is how we tell GHC that this file needs to be run through the `hsx2hs` pre-processor in order to work. So, that options line looks a bit like line noise. You can try to remember it like this:

1. `-F` says we want to filter the source code (or maybe trans*F*orm the source code)
2. `-pgmF` specifies the program we want to do the transformation
3. `hsx2hs` is the preprocessor that converts the `hsx` markup to plain-old `hs`

Next we have some imports:

```
import Control.Applicative          ((<$>))
import Control.Monad.Identity       (Identity(runIdentity))
import Data.String                  (IsString(fromString))
import Data.Text                    (Text)
import HSP
import HSP.Monad                    (HSPT(..))
import Happstack.Server.HSP.HTML
import Happstack.Server.XMLGenT
import Happstack.Server             ( Request(rqMethod), ServerPartT
                                    , askRq, nullConf, simpleHTTP
                                    )
```

Now we can define a function which generates an HTML page:

```
hello :: ServerPartT IO XML
hello = unHSPT $ unXMLGenT
  <html>
   <head>
    <title>Hello, HSP!</title>
   </head>
   <body>
    <h1>Hello HSP!</h1>
    <p>We can insert Haskell expression such as this:
        <% show $ sum [1 .. (10 :: Int)] %></p>
    <p>We can use the ServerPartT monad too.
       Your request method was: <% getMethod %></p>
    <hr/>
    <p>We don't have to escape & or >. Isn't that nice?</p>
    <p>If we want <% "<" %> then we have to do something funny.</p>
    <p>But we don't have to worry about
       escaping <% "<p>a string like this</p>" %></p>
    <p>We can also nest <% <span>like <% "this." %> </span> %></p>
   </body>
  </html>
      where
      getMethod :: XMLGenT (HSPT XML (ServerPartT IO)) String
      getMethod = show . rqMethod <$> askRq

main :: IO ()
main = simpleHTTP nullConf $ hello
```

The first thing we notice is that syntax looks pretty much like normal HTML

syntax. There are a few key differences though:

1. like XML, all tags must be closed
2. like XML, we can use shortags (e.g. `<hr />`)
3. We do not have to escape & and >
4. To embed < we have to do something extra funny

The syntax:

```
<% haskell-expression %>
```

allows us to embed a Haskell expression inside of literal XML.

As shown in this line:

```
<p>We can also nest <% <span>like <% "this." %> </span> %></p>
```

we can freely nest Haskell and XML expressions.

hsx QuasiQuoter

Instead of using the `hsx2hs` preprocessor, we can use the `[hsx| |] QuasiQuoter`. We can take the code from the previous section and make three simple changes.

First we remove the `-F -pgmFhsx` pragma and enable the `QuasiQuotes LANGUAGE` extension instead.

```
{-# LANGUAGE FlexibleContexts, OverlappingInstances, QuasiQuotes #-}
module Main where
```

Next we have some imports:

```
import Control.Applicative          ((<$>))
import Control.Monad.Identity       (Identity(runIdentity))
import Data.String                  (IsString(fromString))
import Data.Text                    (Text)
import HSP
import HSP.Monad                    (HSPT(..))
import Happstack.Server.HSP.HTML
import Happstack.Server.XMLGenT
import Happstack.Server             ( Request(rqMethod), ServerPartT
                                    , askRq, nullConf, simpleHTTP
                                    )
```

The second change is to import the `hsx QuasiQuoter`:

```
import Language.Haskell.HSX.QQ      (hsx)
```

The third change is to wrap the XML markup inside of a `[hsx| |]`:

```
hello :: ServerPartT IO XML
hello = unHSPT $ unXMLGenT
```

```
 [hsx|
  <html>
   <head>
    <title>Hello, HSP!</title>
   </head>
   <body>
    <h1>Hello HSP!</h1>
    <p>We can insert Haskell expression such as this:
        <% show $ sum [1 .. (10 :: Int)] %></p>
    <p>We can use the ServerPartT monad too.
       Your request method was: <% getMethod %></p>
    <hr/>
    <p>We don't have to escape & or >. Isn't that nice?</p>
    <p>If we want <% "<" %> then we have to do something funny.</p>
    <p>But we don't have to worry about
       escaping <% "<p>a string like this</p>" %></p>
    <p>We can also nest <% <span>like <% "this." %> </span> %></p>
   </body>
  </html>
 |]
      where
      getMethod :: XMLGenT (HSPT XML (ServerPartT IO)) String
      getMethod = show . rqMethod <$> askRq
```

The `main` function is unaltered.

```
main :: IO ()
main = simpleHTTP nullConf $ hello
```

As a quick aside – in the `hello` example, the `getMethod` function gives the type checker enough information to infer the type of the XML generated by the `[hsx| |]` quasiquoter. If you comment out the call to `getMethod` you will get an ambiguous type error message. One way to fix this by adding an explicit type signature to the closing html tag like:

```
</html>  :: XMLGenT (HSPT XML (ServerPartT IO)) XML
```

In practice, we often create a page template function, similar to `defaultTemplate` which already contains the `<html>`, `<head>`, and `<body>` tags and also provides the extra type information needed.

The `HSPT` monad itself is covered in a later section.

What do hsx2hs and [hsx| |] actually do?

In order to use HSX it is very useful to understand what is actually going on behind the magic. In these examples we are actually use to separate, but related libraries, the `hsx2hs` library and the `hsp` library.

If we have the line:

```
foo :: XMLGenT (ServerPartT IO) XML
foo = <span class="bar">foo</span>
```

and we run `hsx2hs`, it gets turned into a line like this:

```
foo :: XMLGenT (ServerPartT IO) XML
foo = genElement (Nothing, "span")
        [ asAttr ("class" := "bar") ] [asChild ("foo")]
```

We see that the XML syntax has simply been translated into normal haskell function calls.

The `hsx QuasiQuoter` performs the same transformation as `hsx2hs`.

This is all that `hsx2hs` does. An important thing to note is that `hsx2hs` does not include any functions of those names or even specify their type signatures. The functions come from another library – in this case `hsp`. However, you could implement different behavior by importing a different library.

the XMLGenT type

There are a few types and classes that you will need to be familiar with from the `hsp` library. The first type is the `XMLGenT` monad transformer:

```
newtype XMLGenT m a = XMLGenT (m a)

-- | un-lift.
unXMLGenT :: XMLGenT m a -> m a
unXMLGenT (XMLGenT ma) =  ma
```

This seemingly useless type exists solely to make the type-checker happy. Without it we would need an instance like:

```
instance ( EmbedAsChild (IdentityT m) a
         , Functor m
         , Monad m
         , m ~ n
         ) =>
         EmbedAsChild (IdentityT m) (n a) where
  asChild = ...
```

Unfortunately, because `(n a)` is so vague, that results in overlapping instances which cannot be resolved without `IncohorentInstances`. And, in my experience, enabling `IncohorentInstances` is *never* the right solution.

So, when generating XML you will generally need to apply `unXMLGenT` to the result to remove the `XMLGenT` wrapper as we did in the `hello` function. Anyone who can figure out to do away with the `XMLGenT` class will be my personal hero.

the `XMLGen` class

Next we have the `XMLGen` class:

```
class Monad m => XMLGen m where
 type XMLType m
 type StringType m
 data ChildType m
 data AttributeType m
 genElement    :: Name (StringType m)
               -> [XMLGenT m [AttributeType m]]
               -> [XMLGenT m [ChildType m]]
               -> XMLGenT m (XMLType m)
 genEElement   :: Name (StringType m)
               -> [XMLGenT m [AttributeType m]]
               -> XMLGenT m (XMLType m)
 genEElement n ats = genElement n ats []
 xmlToChild    :: XMLType m    -> ChildType m
 pcdataToChild :: StringType m -> ChildType m
```

You will notice that we have a type-class instead of just simple functions and types. One feature of HSX is that it is not tied to any particular XML representation. Instead, the XML representation is based on the monad we are currently inside. For example, inside of a javascript monad, we might generate javascript code that renders the XML, inside of another monad, we might generate the `Node` type used by the `heist` template library. We will see some examples of this in a later section.

The `data` and `type` declarations appearing inside the class declaration are allowed because of the `TypeFamilies` extension. For a detailed coverage of type families see this wiki entry.

Most of these functions and types are used internally and not used directly by the developer. You will, however, see two of the associated types appear in your type signatures.

the `XMLType m` type synonym

The `XMLGen` type-class defines an associated type synonym `XMLType m`:

```
type XMLType m
```

`XMLType m` is a synonym for whatever the xml type is for the monad `m`. We can write an XML fragment that is parameterized over an arbitrary monad and xml type like this:

```
bar :: (XMLGenerator m) => XMLGenT m (XMLType m)
bar = <span>bar</span>
```

the StringType m type synonym

The `XMLGen` type-class also defines an associated type synonym `StringType m`:

```
type StringType m
```

That is because some types, such as `HSP.XML.XML` are based around `Text`, while others, such as `JMacro` are based around `String`.

the EmbedAsChild class

The `EmbedAsChild` is used to turn a value into a list of children of an element:

```
type GenChildList m      = XMLGenT m [Child m]

-- | Embed values as child nodes of an XML element. The parent type
-- will be clear from the context so it is not mentioned.
class XMLGen m => EmbedAsChild m c where
  asChild :: c -> GenChildList m
```

There are generally many instances of `EmbedAsChild` allowing you to embed `String`, `Text`, `Int`, and other values. You might find it useful to create additional instances for types in your program. We will some some examples later in this tutorial.

To use the `EmbedAsChild` class we us the `<% %>` syntax shown earlier. For example, when we write:

```
a :: (XMLGenerator m) => GenChildList m
a = <% 'a' %>
```

It gets turned into:

```
a :: (XMLGenerator m) => GenChildList m
a = (asChild ('a'))
```

the EmbedAsAttr class

The `EmbedAsAttr` class is similar to the `EmbedAsChild` class. It is used to turn arbitrary values into element attributes.

```
type GenAttributeList m = XMLGenT m [Attribute m]

-- | Similarly embed values as attributes of an XML element.
class XMLGen m => EmbedAsAttr m a where
   asAttr :: a -> GenAttributeList m
```

If we have some attributes like this:

```
foo = <span class="foo" size=(80 :: Int) bogus=False>foo</span>
```

It will get translated to:

```
foo
 = (genElement (Nothing, "span")
       [asAttr ("class" := "foo"), asAttr ("size" := (80 :: Int)),
        asAttr ("bogus" := False)]
       [asChild ("foo")])
```

which might be rendered as:

```
<span class="foo" size="80" bogus="false">foo</span>
```

the `XMLGenerator` class

You may have noticed that some of the examples had a class constraint `(XMLGenerator m)`:

```
bar :: (XMLGenerator m) => XMLGenT m (XMLType m)
bar = <span>bar</span>
```

`XMLGenerator` is just a class alias. It is defined as such:

```
class ( XMLGen m
      , SetAttr m (XMLType m)
      , AppendChild m (XMLType m)
      , EmbedAsChild m (XMLType m)
      , EmbedAsChild m [XMLType m]
      , EmbedAsChild m Text
      , EmbedAsChild m Char -- for overlap purposes
      , EmbedAsChild m ()
      , EmbedAsAttr m (Attr Text Text)
      , EmbedAsAttr m (Attr Text Int)
      , EmbedAsAttr m (Attr Text Bool)
      ) => XMLGenerator m
```

It contains a list of common instances that all xml generation monads are expected to provide. It just saves you from having to list all thoses instances by hand when you use them.

HSPT Monad

The module `HSP.Monad` defines a type:

```
newtype HSPT xml m a = HSPT { unHSPT :: m a }
```

There is an `XMLGenerator` instance for it which can be used to generate `XML`:

```
instance (Functor m, Monad m) => XMLGenerator (HSPT XML m)
```

The `HSPT` type is basically the same as the `IdentityT` type except it has the `phantom` parameter `xml`. This makes it possible to create multiple `XMLGenerator` instances for `HSPT` that generate different xml types.

HSX by Example

First we have a simple function to render the pages and print them to stdout:

```
{-# LANGUAGE FlexibleContexts, QuasiQuotes,
    TypeFamilies, OverloadedStrings #-}
module Main where

import Control.Monad.Identity      (Identity(..))
import           Data.Text.Lazy    (Text)
import qualified Data.Text.Lazy.IO as Text
import Data.String                 (fromString)
import Language.Haskell.HSX.QQ     (hsx)
import HSP
import HSP.Monad
import HSP.HTML4
import Happstack.Server.HSP.HTML (defaultTemplate)

printXML :: HSPT XML Identity XML -> IO ()
printXML = Text.putStrLn . renderAsHTML . runIdentity . unHSPT
```

HSX and `do` syntax

It is possible to use hsx markup inside a `do`-block:

```
doBlock :: (XMLGenerator m, StringType m ~ Text) =>
           XMLGenT m (XMLType m)
doBlock =
   do [hsx| <div>
       <p>A child element</p>
      </div> |]
```

Notice that we indent the closing `</div>` tag. That indentation rule is consistent with the specification for how do-notation works. It is intend for the same reason

that `if .. then .. else ..` blocks have to be idented in a special way inside `do`-blocks.

In newer versions of HSX, this restriction has been lifted.

`defaultTemplate`

There is a bit of boiler plate that appears in ever html document such as the `<html>`, `<head>`, `<title>`, and `<body>`; tags. The `defaultTemplate` function from `happstack-hsp` provides a minimal skeleton template with those tags:

```
module Happstack.Server.HSP.HTML where

defaultTemplate :: ( XMLGenerator m, EmbedAsChild m headers
                   , EmbedAsChild m body, StringType m ~ Text) =>
                   Text      -- ^ text for \<title\> tag
                -> headers   -- ^ extra headers for \<head\> tag.
                             Use @()@ if none.
                -> body      -- ^ content for \<body\> tags.
                -> m (XMLType m)
```

How to embed empty/nothing/zero

`defaultTemplate` requires that we pass in `headers` and a `body`. But what if we don't have any headers that we want to add?

Most `XMLGenerator` monads provide an `EmbedAsChild m ()` instance, such as this one:

```
instance EmbedAsChild (HSPT XML m) () where
    asChild () = return []
```

So, we can just pass in `()` like so:

```
main :: IO ()
main = printXML $ defaultTemplate "empty" () ()
```

Which will render as such:

```
 <html
  ><head
   ><title
     >empty</title
     ></head
   ><body
   ></body
   ></html
 >
```

Creating a list of children

Sometimes we want to create a number of child elements without knowing what their parent element will be. We can do that using the:

```
<%> ... </%>
```

syntax. For example, here we return two paragraphs:

```
twoParagraphs :: (XMLGenerator m, StringType m ~ Text) =>
                 XMLGenT m [ChildType m]
twoParagraphs = [hsx|
  <%>
   <p>Paragraph one</p>
   <p>Paragraph two</p>
  </%>
|]
```

We can embed `twoParagraphs` in parent element using the normal `<% %>` syntax:

```
twoParagraphsWithParent :: (XMLGenerator m, StringType m ~ Text) =>
                           XMLGenT m (XMLType m)
twoParagraphsWithParent = [hsx|
    <div>
     <% twoParagraphs %>
    </div>
|]
```

`if .. then .. else ..`

Using an `if .. then .. else ..` is straight-foward. But what happens when you don't really want an `else` case? This is another place we can use `()`:

```
ifThen :: Bool -> IO ()
ifThen bool =
    printXML $ defaultTemplate "ifThen" () $ [hsx|
     <div>
      <% if bool
         then <%
                <p>Showing this thing.</p>
              %>
        else <% () %>
       %>
     </div> |]
```

Lists of attributes & optional attributes

Normally attributes are added to an element using the normal html attribute syntax. HSX, has a special extension where the last attribute can be a Haskell expression which returns a list of attributes to add to the element. For example:

```
attrList :: IO ()
attrList =
   printXML $ defaultTemplate "attrList" () $ [hsx|
    <div id="somediv" [ "class" := "classy"
                      , "title" := "untitled" :: Attr Text Text
                      ] >
    </div> |]
```

The type of the elements of the list can be anything with an `EmbedAsAttr m a` instance. In this case we create a list of `Attr` values:

```
data Attr n a = n := a
```

We need the type annotation `Attr Text Text` because, due to `OverloadedStrings` the compiler can't automatically determine what type we want for the string literals.

We can use the attribute list feature to conditionally add attributes using a simple `if .. then .. else ..` statment:

```
optAttrList :: Bool -> IO ()
optAttrList bool =
  printXML $ defaultTemplate "attrList" () $ [hsx|
   <div id="somediv" (if bool
                      then [ "class" := "classy"
                           , "title" := "untitled" :: Attr Text Text]
                      else []) >
   </div> |]
```

A clever trick here is to use the list comprehension syntax as an alternative to the `if .. then .. else`:

```
optAttrList2 :: Bool -> IO ()
optAttrList2 bool =
    printXML $ defaultTemplate "attrList" () $ [hsx|
     <div id="somediv"
        [ attr | attr <- [ "class" := "classy"
                         , "title" := "untitled" :: Attr Text Text]
        , bool] >
    </div> |]
```

this trick works better when you are only attempting to add a single extra attribute:

```
optAttrList3 :: Bool -> IO ()
optAttrList3 bool =
    printXML $ defaultTemplate "attrList" () $ [hsx|
     <div id="somediv"
       [ "class" := "classy" :: Attr Text Text | bool] >
     </div> |]
```

Source code for the app is here.

HSX and compilation errors

One drawback to HSX is that it can result in some pretty ugly (and sometimes very long) error messages. Fortunately, the errors are almost always the same type of thing, so after a little experience it is easy to see what is going wrong. Here are some tips if you run into errors:

`hsx2hs` line numbers are usually wrong

As we saw, `hsx2hs` transforms the literal XML into normal Haskell code. Unfortunately, the error positions reported by GHC reflect where the error occurred in the transformed code, not the original input. `hsx2hs` tries to help GHC by inserting `LINE` pragmas. While that helps to a degree, it still leaves a fair bit of fuzz.

The trick is to look towards the bottom of the error message where it will usually show you the expression that contained the error. For example, if we have:

```
typeError :: (XMLGenerator m, StringType m ~ Text) =>
             XMLGenT m (XMLType m)
typeError = [hsx| <foo><% 1 + 'a' %></foo> |]
```

We will get an error like:

```
Templates/HSX/What.lhs:456:20:
Could not deduce (Num Char) arising from a use of `+'
from the context (XMLGenerator m, StringType m ~ Text)
  bound by the type signature for
     typeError :: (XMLGenerator m, StringType m ~ Text) =>
                  XMLGenT m (XMLType m)
  at Templates/HSX/What.lhs:455:16-77
Possible fix: add an instance declaration for (Num Char)
In the first argument of `asChild', namely `(1 + 'a')'
In the first argument of `asChild', namely `((asChild (1 + 'a')))'
In the expression: asChild ((asChild (1 + 'a')))
```

The last line says:

```
    In the expression: asChild ((asChild (1 + 'a')))
```

And the sub-expresion `1 + 'a'` is, indeed, where the type error is.

A bug report about the line number issue has been filed, and there are ideas on how to fix it. You can read more here.

Using the new hsx quasi-quoters helps significantly with the accuracy of line numbers.

Note on Next Two Sections

The errors describe in the next section do not happen anymore due to improvements to HSX. However, similar errors can arise so they are still instructive even though they are a bit out of date.

Overlapping Instances

Another common error is that of overlapping instances. For example, if we wrote the following:

```
overlapping = [hsx| <p>overlapping</p> |]
```

We would get an error like:

```
TemplatesHSP.markdown.lhs:495:36:
    Overlapping instances for EmbedAsChild m0 [Char]
      arising from a use of ‘asChild’
    Matching instances:
      instance [overlap ok] XMLTypeGen m => EmbedAsChild m String
        -- Defined in ‘HSX.XMLGenerator’
      instance EmbedAsChild Identity String -- Defined in ‘HSP.Identity’
      instance Monad m => EmbedAsChild (ServerPartT m) String
        -- Defined in ‘HSP.ServerPartT’
    (The choice depends on the instantiation of ‘m0’
     To pick the first instance above, use -XIncoherentInstances
     when compiling the other instance declarations)
    In the expression: asChild ("overlapping")
    In the third argument of ‘genElement’, namely
      ‘[asChild ("overlapping")]’
    In the expression:
      (genElement (Nothing, "p") [] [asChild ("overlapping")])
```

I have never enabled `IncoherentInstances` and actually had it do what I wanted. In this case, the solution is to add an explicit type signature that mentions the missing constraint:

```
{-
overlapping :: (EmbedAsChild m String) => XMLGenT m (XMLType m)
```

```
overlapping = <p>overlapping</p>
-}
```

In general, there can be a lot of required `EmbedAsChild` and `EmbedAsAttr` instances. So, often times you can save a lot of typing by using the `XMLGenerator` class alias:

```
{-
overlapping' :: (XMLGenerator m) => XMLGenT m (XMLType m)
overlapping' = <p>overlapping</p>
-}
```

Ambiguous Types

Sometimes a type signature for the parent function is not enough. For example, let's say we have:

```
ambiguous :: (EmbedAsChild m String, StringType m ~ Text) =>
             XMLGenT m (XMLType m)
ambiguous = [hsx| <p><% fromString "ambiguous" %></p> |]
```

That will generate an error like this one:

```
TemplatesHSP.markdown.lhs:557:28:
    Ambiguous type variable ‘c0’ in the constraints:
      (IsString c0)
        arising from a use of ‘fromString’
        at TemplatesHSP.markdown.lhs:557:28-37
      (EmbedAsChild m c0)
        arising from a use of ‘asChild’
        at TemplatesHSP.markdown.lhs:557:19-25
    Probable fix: add a type signature that fixes these type variable(s)
    In the first argument of ‘asChild’, namely
      ‘(fromString "ambiguous")’
    In the first argument of ‘asChild’, namely
      ‘((asChild (fromString "ambiguous")))’
    In the expression: asChild ((asChild (fromString "ambiguous")))
Failed, modules loaded: none.
```

Here we are trying to use `fromString` to convert `"ambiguous"` into some type, and then we embed that type using `asChild`. But there is not enough information to figure out what the intermediate type should be. It is the same problem we have if we try to write:

```
\str -> show (read str)
```

The solution here is to add an explicit type signature to the result of `fromString`:

```
{-
ambiguous :: (EmbedAsChild m Text) => XMLGenT m (XMLType m)
ambiguous = <p><% (fromString "ambiguous") :: Text %></p>
-}
```

HSP and internationalization (aka, i18n)

You will need to install `happstack-hsp` and `shakespeare-i18n` for this section.

Internationalization (abbreviated to the numeronym i18n) and localization (L10n) generally refer to the processing of making an application usuable by people that speak different languages, use different alphabets and keyboards, and have different conventions for things like formatting times and dates, currency, etc.

Proper handling of these issues can run deep into your code. For example, English speakers often think of people as having a first name and a last name – but when you look at how people's names are used around the world, you realize these familiar terms are not universally applicable. So, a type like:

```
data Name = Name { firstName :: Text, lastNime :: Text }
```

may not be sufficient.

The haskell wiki lists a bunch of methods for translating strings into multiple languages.

In this example, we show how we can use native haskell types datas, a translator friendly file format, and HSP to do some simple internationalization. We will build on top of the `shakespeare-i18n` library.

As usual, we start off with a bunch of imports and pragmas:

```
{-# LANGUAGE FlexibleContexts, FlexibleInstances, TemplateHaskell,
    MultiParamTypeClasses, OverloadedStrings, QuasiQuotes,
    TypeFamilies #-}
module Main where

import Control.Applicative   ((<$>))
import Control.Monad         (msum)
```

```
import Control.Monad.Reader  (ReaderT, ask, runReaderT)
import Control.Monad.Trans   (MonadIO(liftIO))
import Data.Map              (Map, fromList)
import qualified Data.Map    as Map
import Data.Monoid           ((<>))
import qualified Data.Text   as Strict
import qualified Data.Text.Lazy as Lazy
import Happstack.Server      ( ServerPart, ServerPartT, dir
                             , lookTexts', mapServerPartT
                             , nullConf, nullDir, queryString
                             , simpleHTTP , acceptLanguage
                             , bestLanguage
                             )
import Happstack.Server.HSP.HTML
import Happstack.Server.XMLGenT
import HSP
import HSP.Monad             (HSPT(..))
import Language.Haskell.HSX.QQ (hsx)
import Text.Shakespeare.I18N ( RenderMessage(..), Lang, mkMessage
                             , mkMessageFor, mkMessageVariant)
import System.Random (randomRIO)
```

HSP + i18n Core Concept

Instead of using strings directly in our templates we could create a data type where each constructor represents a phrase, sentence, or paragraph that we want to put on the page. For example, we could define the type:

```
data Message = Hello | Goodbye
```

Then we could provide a translation function for each language we support:

```
translation_en :: Message -> Strict.Text
translation_en Hello       = "hello"
translation_en Goodbye     = "goodbye"

translation_lojban :: Message -> Strict.Text
translation_lojban Hello   = "coi"
translation_lojban Goodbye = "co'o"

translations :: Map Strict.Text (Message -> Strict.Text)
translations =
    fromList [ ("en"    , translation_en)
             , ("lojban", translation_lojban)
             ]
```

```
translate :: Strict.Text -> Message -> Strict.Text
translate lang msg =
    case Map.lookup lang translations of
      Nothing           -> "missing translation"
      (Just translator) ->
          translator msg
```

and then in our templates we can write:

```
helloPage :: ( XMLGenerator m
             , EmbedAsChild m Strict.Text
             , StringType m ~ Lazy.Text
             ) =>
            Strict.Text -> XMLGenT m (XMLType m)
helloPage lang = [hsx|
    <html>
     <head>
      <title><% translate lang Hello %></title>
     </head>
     <body>
      <p><% translate lang Hello %></p>
     </body>
    </html>
    |]
```

The principle behind this approach is nice, but in practice, it has a few problems:

1. having to write the translation functions in the Haskell source is not a very friendly format for the people who will be doing the translations.
2. having to call 'translate' explicitly is boring, tedious, and error prone
3. having to pass around the desired 'lang' manually is also boring, tedious, and error prone

Fortunately, we can work around all these issues quite simply.

the RenderMessage class

`shakespeare-i18n` provides a simple class for providing translations:

```
type Lang = Text

class RenderMessage master message where
  renderMessage :: master     -- ^ translation variant
                -> [Lang]     -- ^ desired languages in descending
                              --   order of preference
                -> message    -- ^ message we want translated
                -> Text       -- ^ best matching translation
```

`renderMessage` is pretty straight-forward. It takes a list of preferred languages and a message datatype (such as `Message` type we defined above) and returns the best matching translation. The only mysterious part is the `master` argument. (Personally, I think `variant` might be a better name for the argument). The argument exists so that you can provide more than one set of translations for the same message type.

For example, let's say that we had defined the `Message` type in a library. Being the nice people we are, we also provide a set of translations for the `Message` type. However, someone using our library may want to provide a completely different set of translations that are more appropriate to their application. For example, in the library we might have:

```
data LibraryI18N = LibraryI18N

instance RenderMessage LibraryI18N Message where
    renderMessage = ...
```

But the user could provide their own translations for `Message` via:

```
data AppI18N = AppI18N

instance RenderMessage AppI18N Message where
    renderMessage = ...
```

shakespeare-i18n translation files

Writing the translations in your Haskell source can be pretty inconvenient. Especially if you are working with a team of outsourced translators. Fortunately, `shakespeare-i18n` has support for external translation files.

To keep things simple:

1. each language will have its own translation file
2. the file will be named *lang*`.msg` where `lang` is a language code such as `en`, `en-GB`, `fr`, etc
3. the translation files will all be in a subdirectory which contains nothing but translations
4. the `.msg` files must be UTF-8 encoded

So for this example we will have three files:

```
messages/standard/en.msg
messages/standard/en-GB.msg
messages/standard/jbo.msg
```

- `en.msg` is a set of generic English translations.

- `en-GB.msg` is a set of English translations using spellings and idioms common to Great Britain
- `jbo.msg` is a set of Lojban translations

The contents of the files are:

`messages/standard/en.msg`

```
Hello: greetings
Goodbye: seeya
Problems n@Int thing@Thing: Got #{show n} #{plural_en n "problem" "problems" } but a #{thing_tr '
```

`messages/standard/en-GB.msg`

```
Hello: all right?
Goodbye: cheerio
Problems n thing: Got #{show n} #{plural_en n "problem" "problems" } but a #{thing_tr "en-gb" thi
```

`messages/standard/jbo.msg`

```
Hello: coi
Goodbye: co'o
```

The format is very simple. Each line looks like:

```
Constructor arg0 arg1 .. argn: translation text
```

1. `Constructor` is a valid Haskell constructor name that we will use to reference this translation
2. it is followed by 0 or more variable names
3. then there is a `:`
4. and then there is the translation

You may also notice that in `en.msg` the arguments contain types like `n@Int`. And some of translations contain markup like `#{show n}`. You can probably guess what those things mean – we will come back to them shortly.

You may also notice that the Lojban translation is missing the `Problems` constructor. Since there is no translation provided, `renderMessage` will use the default translation (which, in this case will come from `en.msg`).

Due to TH staging restrictions this code must come before the `mkMessage` call below. But we are not ready to talk about it yet in the tutorial. So ignore it until later.

```
plural_en :: (Integral i) => i -> String -> String -> String
plural_en 1 x _ = x
plural_en _ _ y = y

data Thing = TypeError | SegFault deriving (Enum, Bounded, Show)

mkMessageFor "DemoApp" "Thing" "messages/thing" ("en")
```

```
thing_tr :: Lang -> Thing -> Strict.Text
thing_tr lang thing = renderMessage DemoApp [lang] thing
```

To load the message files we first need to define our `master` type:

```
data DemoApp = DemoApp
```

Then we just call `mkMessage`:

```
mkMessage  "DemoApp" "messages/standard" ("en")
```

`mkMessage` is a Template Haskell function which:

1. reads the `.msg` files
2. creates a new datatype based on the constructors it found
3. creates a `RenderMessage` instance

`mkMessage` has the following type:

```
mkMessage :: String     -- ^ name of master translation type
          -> FilePath   -- ^ path to folder which contains the `.msg` files
          -> Lang       -- ^ default language
          -> Q [Dec]
```

If we use `-ddump-splices` we see that the `mkMessages` call above generated the following for us:

```
data DemoAppMessage
    = MsgHello
    | MsgGoodbye
    | MsgProblems { translationsMessageN     :: Int
                  , translationsMessageThing :: Thing
                  }


instance RenderMessage DemoApp DemoAppMessage where
    renderMessage = ...
```

It has created a new type for us `DemoAppMessage` where each constructor is derived from the constructors found in the `en.msg` file. The constructor names all have the prefix `Msg`. That is just to avoid name collisions with the other constructors in your application.

It has also created a `RenderMessage` instance with all the translations (not shown for the sake of readability).

Now we can do:

```
*Main> renderMessage DemoApp ["en"] MsgHello
"greetings"
```

Note that because the message files are read in using Template Haskell at compile time, we do not need to install them on the live server. Also, if you change the

`.msg` files, you will not see the changes until you recompile.

Constructor arguments, #{ }, and plurals

The `Problems` constructor in the `en.msg` file appears considerably more complicate than the `Hello` and `Goodbye` cases:

```
Problems n@Int thing@Thing: Got #{show n} #{plural_en n "problem" "problems" } but a #{thing_tr '
```

There are a few things going on here.

Type Annotations

The `Problems` constructor takes two arguments: `n` and `thing`. In order to create the `MsgProblems` constructor, `mkMessage` needs to know the types of those arguments. So, we add the type annotations using the `@` syntax. We only need the type annotations in the default translation file. The default translation file is specified as the third argument to `mkMessage` – which in this example is `"en"`.

The types of the arguments can be any valid Haskell type. In this case 'Int' and 'Thing'. 'Thing' is just a normal Haskell datatype which we will define right now as:

```
data Thing = TypeError | SegFault deriving (Enum, Bounded, Show)
```

Variable Splices

The `#{ }` syntax allows you to call a Haskell function and splice the result into the message. For example:

```
 #{show n}
```

will convert `n` to a `String` and splice the `String` into the message. The expression inside the `#{ }` must be a pure expression and it must have a type that is an instance of the `ToMessage` class:

```
class ToMessage a where
  toMessage :: a -> Text
```

By default, only `String` and `Text` have `ToMessage` instances.

Remember that `mkMessage` generates code which gets spliced into the current module. That means the code inside `#{ }` has access to any functions and types which are available in the module that calls `mkMessage`.

Handling plurals and other language specifics

In English, we say:

- I have 1 problem
- I have 0 problems
- I have 10 problems

In our translations, we don't want to say *I have 1 problem(s).* We can handle this pluralization issue by creating a simple helper function such as this one:

```
plural_en :: (Integral i) => i -> String -> String -> String
plural_en 1 x _ = x
plural_en _ _ y = y
```

Looking at `en.msg` you notice that we need to use `plural_en` twice to make the grammar sound natural. When creating messages is good to use whole phrases and sentences because changes in one part of a sentence can affect other parts of the sentence. Rules about plurals, word order, gender agreement, etc, vary widely from one language to the next. So it is best to assume as little as possible and give the translators as much flexibility as possible.

Translating Existing Types

`mkMessage` creates a new type from the constructors it finds in the `.msg` files. But sometimes we want to create a translation for an existing type. For example, we need to translate the `Thing` type. We can do that by creating a function like:

```
thing_tr :: Lang -> Thing -> Text
```

Which we can call in the translation file like:

```
 #{thing_tr "en" thing}
```

But, how do we implement `thing_tr`? One option is to simply write a function like:

```
thing_tr :: Lang -> Thing -> Text
thing_tr lang TypeError | lang == "en" = "type error"
thing_tr lang SegFault  | lang == "en" = "segmentation fault"
thing_tr _    thing     = thing_tr "en" thing
```

But, now someone has to update the Haskell code to add new translations. It would be nice if all the translations came from `.msg` files.

The `mkMessageFor` function allows us to create translations for an existing type:

```
mkMessageFor ::
     String    -- ^ master type
  -> String    -- ^ data to translate
  -> FilePath  -- ^ path to `.msg` files
```

```
  -> Lang       -- ^ default language
  -> Q [Dec]
```

We can create a set of `.msg` files for the `Thing` type like this (note the file path):

`messages/thing/en.msg`

```
TypeError: type error
SegFault: seg fault
```

And then use `mkMessageFor` to create a `RenderMessage` instance:

```
mkMessageFor "DemoApp" "Thing" "messages/thing" "en"
```

That will create this instance for us:

```
-- autogenerated by `mkMessageFor`
instance RenderMessage DemoApp Thing where
    renderMessage = ...
```

Because `mkMessageFor` is creating a `RenderMessage` for an existing type, it does not need to append `Message` to the type name or prefix the constructors with `Msg`. Now we can define our `thing_tr` function like this:

```
thing_tr :: Lang -> Thing -> Text
thing_tr lang thing = renderMessage DemoApp [lang] thing
```

This is definitely a bit roundabout, but it is the best solution I can see using the existing `shakespeare-i18n` implementation.

Alternative Translations

We can use `mkMessageVariant` to create an alternative set of translations for a type that was created by `mkMessage`. For example:

```
data DemoAppAlt = DemoAppAlt

mkMessageVariant "DemoAppAlt" "DemoApp" "messages/alt" "en"
```

Using messages in `HSX` templates

To use the `DemoAppMessage` type in an `HSX` template, all we need is an `EmbedAsChild` instance.

The instance will need to know what the client's preferred languages are. We can provide that by putting the users language preferences in a `ReaderT` monad:

```
type I18N  = HSPT XML (ServerPartT (ReaderT [Lang] IO))
```

Next we create the `EmbedAsChild` instance:

```
instance EmbedAsChild I18N DemoAppMessage where
    asChild msg =
        do lang <- ask
           asChild $ Lazy.fromStrict $ renderMessage DemoApp lang msg
```

Now we can use the message constructors inside our templates:

```
pageTemplate :: (EmbedAsChild I18N body) =>
                Lazy.Text -> body -> I18N XML
pageTemplate title body =
    defaultTemplate title () [hsx|
     <div>
      <% body %>
      <ul>
       <% mapM (\lang ->
           <li>
            <a ["href" := ("?_LANG="<> lang) :: Attr Lazy.Text Lazy.Text]>
             <% lang %>
            </a>
           </li>)
           (["en", "en-GB", "jbo"]) %>
      </ul>
     </div> |]

homePage :: I18N XML
homePage =
   pageTemplate "home"
       [hsx| <p><% MsgHello %></p> |]

goodbyePage :: I18N XML
goodbyePage =
    pageTemplate "goodbye"
        [hsx| <p><% MsgGoodbye %></p> |]

problemsPage :: Int -> Thing -> I18N XML
problemsPage n thing =
    pageTemplate "problems"
        [hsx| <p><% MsgProblems n thing %></p> |]
```

Instead of putting text in the `<p> </p>` tags we just use our message constructors.

Getting the language preferences from `ReaderT [Lang]` is just one possibility. Your application may already have a place to store session data that you can get the preferences from, or you might just stick the preferences in a cookie.

Detecting the preferred languages

The `Accept-Language` header is sent by the client and, in theory, specifies what languages the client prefers, and how much they prefer each one. So, in the absence of any additional information, the `Accept-Language` header is a good starting place. You can retrieve and parse the `Accept-Language` header using the `acceptLanguage` function and then sort the preferences in descending order using `bestLanguage`:

```
acceptLanguage :: (Happstack m) => m [(Text, Maybe Double)]
bestLanguage   :: [(Text, Maybe Double)] -> [Text]
```

You should not assume that the `Accept-Language` header is always correct. It is best to allow the user a way to override the `Accept-Language` header. That override could be stored in their user account, session data, a cookie, etc. In this example we will just use a `QUERY_STRING` parameter `_LANG` to override the `Accept-Language` header.

We can wrap this all up in a little function that converts our `I18N` part into a normal `ServerPart`:

```
withI18N :: I18N a -> ServerPart a
withI18N part = do
  langsOverride <- queryString $ lookTexts' "_LANG"
  langs         <- bestLanguage <$> acceptLanguage
  mapServerPartT (flip runReaderT (langsOverride ++ langs)) (unHSPT part)
```

And finally, we just have our `route` table and `main` function:

```
routes :: I18N XML
routes =
    msum [ do nullDir
              homePage
         , dir "goodbye"  $ goodbyePage
         , dir "problems" $
             do n     <- liftIO $ randomRIO (1, 99)
                let things = [TypeError .. SegFault]
                index <- liftIO $ randomRIO (0, length things - 1)
                let thing  = things !! index
                problemsPage n thing
         ]


main :: IO ()
main = simpleHTTP nullConf $ withI18N routes
```

Source code for the app is here. You will also need to download and unzip the message files here.

Conclusions

In this section we showed how to use `HSX` and `Happstack.Server.I18N`, and `shakespeare-i18n` together to provide an i18n solution. However, there are no dependencies between those libraries and modules. So, you can use other solutions to provide translations for `HSX`, or you can use `shakespeare-i18n` with other template systems.

One thing that would make `shakespeare-i18n` better is a utility to help keep the `.msg` files up-to-date. I have describe my ideas for a tool here. We just need a volunteer to implement it.

JavaScript via JMacro

To use `JMacro` with `happstack` and `hsx`, you should install the `hsx-jmacro` and `happstack-jmacro` packages.

JMacro is a library that makes it easy to include javascript in your templates.

The syntax used by `JMacro` is almost identical to `JavaScript`. So, you do not have to learn some special DSL to use it. In fact, `JMacro` can work with most `JavaScript` you find in the wild. Using `JMacro` has a number of advantages over just using plain-old `JavaScript`.

- syntax checking ensures that your `JavaScript` is syntactically valid at compile time. That eliminates many common `JavaScript` errors and reduces development time.
- hygienic names and scoping automatically and transparently ensure that blocks of `JavaScript` code do not accidentally create variables and functions with conflicting names.
- Antiquotation, marshalling, and shared scope make it easy to splice Haskell values into the `JavaScript` code. It also makes it easy to programmatically generate `JavaScript` code.

The `hsx-jmacro` and `happstack-jmacro` libraries makes it easy to use JMacro with Happstack and `HSP`.

The following examples demonstrate the basics of `JMacro` and how it interfaces with `HSP` and `Happstack`. The examples are intended to demonstrate what is possible with `JMacro`. The examples are not intended to demonstrate good `JavaScript` practices. For example, many developers frown on the use of the `onclick` attribute in html, or having `<script>` tags in the `<body>`.

The `JMacro` library does not require any external pre-processors. Instead it uses the magic of QuasiQuotation.

`QuasiQuotes` can be enabled via the `LANGUAGE` extension:

```
{-# LANGUAGE CPP, FlexibleInstances, GeneralizedNewtypeDeriving,
    TypeSynonymInstances, QuasiQuotes #-}
```

At this time it is not possible to nest the `JMacro` quasiquoter inside the `hsx` quasiquoter. However, we can work around this by using the `hsx2hs` preprocessor:

```
{-# OPTIONS_GHC -F -pgmFhsx2hs #-}
```

Next we have a boatload of imports. Not all of these are required to use `JMacro`. Many are just used for the demos.

There is one really import thing to note though. If you look at the import for `Language.Javascript.JMacro`, you will find that there are a bunch of things imported like `jsVarTy` which we never call explicitly in this demo. The calls to these functions are generated automatically by the `JMacro` quasi-quoters. `JMacro` can not automatically add these imports, so you will need to do it by hand if you use explicit import lists. Alternatively, you can just import `Language.Javascript.JMacro` without an explicit import list.

```
import Control.Applicative ((<$>), optional)
import Control.Monad       (msum)
import Control.Monad.State (StateT, evalStateT)
import Control.Monad.Trans (liftIO)
import qualified Data.Map  as Map
import Data.Maybe          (fromMaybe)
import Data.String         (fromString)
import Happstack.Server    ( Response, ServerPartT, dir
                           , mapServerPartT, look
                           , nullConf, ok, simpleHTTP
                           , toResponse)
import Happstack.Server.HSP.HTML  (defaultTemplate)
import Happstack.Server.JMacro    (jmResponse)
import HSP
import HSP.Monad                  (HSPT(..))
import Happstack.Server.XMLGenT   () -- Happstack instances
                                     -- for XMLGenT and HSPT
import HSP.JMacro                 ( IntegerSupply(..)
                                  , nextInteger')
import Language.Javascript.JMacro ( ToJExpr(..), Ident(..)
                                  , JStat(..), JExpr(..)
                                  , JVal(..), jmacro, jsv
                                  , jLam, jVarTy)
import System.Random              (Random(..))
```

In order to ensure that each `<script>` tag generates unique variables names, we need a source of unique prefixes. An easy way to do that is to wrap the `ServerPartT` monad around a `StateT` monad that supplies integers:

```
type JMacroPart = HSPT XML (ServerPartT (StateT Integer IO))

instance IntegerSupply JMacroPart where
  nextInteger = nextInteger'
```

The `nextInteger'` helper function has the type:

```
nextInteger' :: (MonadState Integer m) => m Integer
```

To use JMacroPart with `simpleHTTP`, we just evaluate the `StateT` monad:

```
main :: IO ()
main = simpleHTTP nullConf $ flatten handlers
    where
      flatten :: JMacroPart a -> ServerPartT IO a
      flatten = mapServerPartT (flip evalStateT 0) . unHSPT
```

JMacro in a <script> tag

Now that we have the scene set, we can actually look at some JMacro usage.

In this example we embed a single JavaScript block inside the page:

```
helloJMacro :: JMacroPart Response
helloJMacro =
  toResponse <$> defaultTemplate (fromString "Hello JMacro") ()
    <div>
     <% [jmacro|
      var helloNode = document.createElement('h1');
      helloNode.appendChild(document.createTextNode("Hello, JMacro!"));
      document.body.appendChild(helloNode);
      |] %>
    </div>
```

We do not need to specify the `<script>` tag explicitly, it will automatically be created for us.

The syntax `[jmacro| ... |]` is the magic incantation for running the `jmacro` quasiquoter. In GHC 7.x, the $ is no longer required, so in theory you could write, `[jmacro| ... |]`. However, HSX has not been updated to support the $ free syntax. So, for now you will need to stick with the $ syntax, despite the compiler warnings saying, `Warning: Deprecated syntax: quasiquotes no longer need a dollar sign: $jmacro`.

JMacro in an HTML attribute (`onclick`, etc)

We can also use JMacro inside html attributes, such as `onclick`.

```
helloAttr :: JMacroPart Response
helloAttr =
 toResponse <$> defaultTemplate (fromString "Hello Attr") ()
```

```
 <h1 style="cursor:pointer"
     onclick=[jmacro| alert("that </tickles>!") |] >Click me!</h1>
```

Note that we do not have to worry about escaping the ", < or > in the onclick handler. It is taken care of for us automatically! The code is automatically escaped as:

onclick="alert("that </tickles>!");"

Automatic escaping of `</`

According to the HTML spec it is invalid for `</` to appear anywhere inside the `<script>` tag.

The JMacro embedding also takes care of handling `</` appearing in string literals. So we can just write this:

```
helloEndTag :: JMacroPart Response
helloEndTag =
    toResponse <$> defaultTemplate (fromString "Hello End Tag") ()
    <%>
     <h1>Tricky End Tag</h1>
     <% [jmacro| alert("this </script> won't mess things up!") |] %>
    </%>
```

And it will generate:

```
<script type="text/javascript">
 alert("this <\/script>; won't mess things up!");
</script>
```

Hygienic Variable Names

So far, using HSP with JMacro looks almost exactly like using HSP with plain-old JavaScript. That's actually pretty exciting. It means that the mental tax for using JMacro over straight JavaScript is very low.

Now let's look at an example of hygienic naming. Let's say we write the following block of JavaScript code:

```
clickMe :: JStat
clickMe =
    [jmacro|

    var clickNode = document.createElement('p');
    clickNode.appendChild(document.createTextNode("Click me!"));
    document.body.appendChild(clickNode);
    var clickCnt = 0;
    clickNode.setAttribute('style', 'cursor: pointer');
    clickNode.onclick = function () {
```

```
      clickCnt++;
      alert ('Been clicked ' + clickCnt + ' time(s).');
    };
|]
```

That block of code tracks how many times you have clicked on the `Click me!` text. It uses a global variable to keep track of the number of clicks. Normally that would spell trouble. If we tried to use that code twice on the same page, both copies would end up writing to the same global variable `clickCnt`.

But, JMacro automatically renames the variables for us so that the names are unique. In the following code each `Click me!` tracks its counts separately:

```
clickPart :: JMacroPart Response
clickPart =
    toResponse <$> defaultTemplate (fromString "Hygienic Naming") ()
                   <div>
                    <h1>A Demo of Happstack+HSP+JMacro</h1>
                    <% clickMe %>
                    <% clickMe %>
                   </div>
```

Non-Hygienic Variable Names

Of course, sometimes we want the code blocks to share a global variable. We can easily do that by changing the line:

```
var clickCnt = 0;
```

to

```
var !clickCnt = 0;
```

The use of ! when declaring a variable disables hygienic naming. Now all the copies of `clickMe2` will share the same counter:

```
clickMe2Init :: JStat
clickMe2Init =
    [jmacro| var !clickCnt = 0; |];

clickMe2 :: JStat
clickMe2 =
    [jmacro|

    var clickNode = document.createElement('p');
    clickNode.appendChild(document.createTextNode("Click me!"));
    document.body.appendChild(clickNode);
    clickNode.setAttribute("style", "cursor: pointer");
    clickNode.onclick = function () {
```

```
      clickCnt++;
      alert ('Been clicked ' + clickCnt + ' time(s).');
     };
    |]

clickPart2 :: JMacroPart Response
clickPart2 =
    toResponse <$> defaultTemplate (fromString "Hygienic Naming")
                   <% clickMe2Init %>
                   <div>
                    <h1>A Demo of Happstack+HSP+JMacro</h1>
                    <% clickMe2 %>
                    <% clickMe2 %>
                   </div>
```

Declaring Functions

Hygienic naming affects function declarations as well. If we want to define a function in `<head>`, but call the function from the `<body>`, then we need to disable hygienic naming. We can do that using the ! trick again:

```
function !hello(noun) { alert('hello ' + noun); }
```

JMacro also has some syntax extensions for declaring functions. We can create an anonymous function using Haskell-like syntax assign it to a variable:

```
var !helloAgain = \noun ->alert('hello again, ' + noun);
```

Another option is to use the ML-like `fun` keyword to declare a function. When using `fun` we do not need the !.

```
fun goodbye noun { alert('goodbye ' + noun); }
```

Or we can do both:

```
fun goodbyeAgain noun -> alert('goodbye again, ' + noun);
```

Here they all are in an example:

```
functionNames :: JMacroPart Response
functionNames =
    toResponse <$> defaultTemplate (fromString "Function Names")
      <% [jmacro|
           function !hello(noun) { alert('hello, ' + noun); }
           var !helloAgain = \noun ->alert('hello again, ' + noun);
           fun goodbye noun { alert('goodbye ' + noun); }
           fun goodbyeAgain noun -> alert('goodbye again, ' + noun);
         |]
```

```
      %>
     <%>
       <button onclick=[jmacro| hello('world'); |]>
         hello
       </button>
       <button onclick=[jmacro| helloAgain('world'); |]>
         helloAgain
       </button>
       <button onclick=[jmacro| goodbye('world'); |]>
         goodbye
       </button>
       <button onclick=[jmacro| goodbyeAgain('world'); |]>
         goodbyeAgain
       </button>
     </%>
```

Splicing Haskell Values into JavaScript (Antiquotation)

We can also splice Haskell values into the JavaScript code by using `( )`. In the following example, the `onclick` action for the `<button>` calls `revealFortune()`. The argument to `revealForture` is the `String` returned by evaluating the Haskell expression `fortunes !! n`.

```
fortunePart :: JMacroPart Response
fortunePart = do
  let fortunes =
       ["You will be cursed to write Java for the rest of your days."
       , "Fortune smiles upon you, your future will be filled with lambdas."
       ]
  n <- liftIO $ randomRIO (0, (length fortunes) - 1)

  toResponse <$> defaultTemplate (fromString "Fortune")
       <% [jmacro|
            fun revealFortune fortune
            {
             var b = document.getElementById("button");
             b.setAttribute('disabled', 'disabled');
             var p = document.getElementById("fortune");
             p.appendChild(document.createTextNode(fortune));
            }
           |]
       %>
       <div>
```

```
        <h1>Your Fortune</h1>
        <p id="fortune">
        <button id="button"
           onclick=[jmacro| revealFortune(`(fortunes !! n)`); |]>
          Click to reveal your fortune
        </button>
        </p>
       </div>
```

Using `ToJExpr` to convert Haskell values to JavaScript

JMacro can embed common types such as `Int`, `Bool`, `Char`, `String`, etc, by default. But we can also embed other types by creating a `ToJExpr` instance for them. For example, let's say we create some types for reporting the weather:

```
data Skies = Cloudy | Clear
    deriving (Bounded, Enum, Eq, Ord, Read, Show)

newtype Fahrenheit = Fahrenheit Double
    deriving (Num, Enum, Eq, Ord, Read, Show, ToJExpr, Random)

data Weather = Weather
    { skies :: Skies
    , temp  :: Fahrenheit
    }
    deriving (Eq, Ord, Read, Show)

instance Random Skies where
    randomR (lo, hi) g =
       case randomR (fromEnum lo, fromEnum hi) g of
         (c, g') -> (toEnum c, g')
    random g = randomR (minBound, maxBound) g

instance Random Weather where
    randomR (Weather skiesLo tempLo, Weather skiesHi tempHi) g =
        let (skies, g') = randomR (skiesLo, skiesHi) g
            (temp, g'') = randomR (tempLo, tempHi) g'
        in ((Weather skies temp), g'')
    random g =
        let (skies, g') = random g
            (temp, g'') = random g'
        in ((Weather skies temp), g'')
```

To pass these values into the generated JavaScript, we simply create a `ToJExpr` instance:

```
class ToJExpr a where
  toJExpr :: a -> JExpr
```

For `Fahrenheit`, we were actually able to derive the `ToJExpr` instance automatically (aka, `deriving (ToJExpr)`), because it is a `newtype` wrapper around `Double` which already has a `ToExpr` instance.

For `Skies`, we can just convert the constructors into JavaScript strings:

```
instance ToJExpr Skies where
    toJExpr = toJExpr . show
```

For the `Weather` type, we create a JavaScript object/hash/associative array/record/whatever you want to call it:

```
instance ToJExpr Weather where
   toJExpr (Weather skies temp) =
       toJExpr (Map.fromList [ ("skies", toJExpr skies)
                             , ("temp",  toJExpr temp)
                             ])
```

Now we can splice a random weather report into our JavaScript:

```
weatherPart :: JMacroPart Response
weatherPart = do
  weather <- liftIO $ randomRIO ((Weather minBound (-40)),
                                 (Weather maxBound 100))
  toResponse <$> defaultTemplate (fromString "Weather Report") ()
        <div>
         <% [jmacro|
             var w = `(weather)`;
             var p = document.createElement('p');
             p.appendChild(document.createTextNode(
                "The skies will be " + w.skies +
                " and the temperature will be " +
                w.temp.toFixed(1) + "F"));
             document.body.appendChild(p);
             |] %>
        </div>
```

`ToJExpr` has an instance for `JSValue` from the `json` library. So, if your type already has a `JSON` istance, you can trivially create a `ToJExpr` instance for it:

```
instance ToJExpr Foo where
  toJExpr = toJExpr . showJSON
```

Using JMacro in external `.js` scripts

So far we have used JMacro to generate JavaScript that is embedded in HTML. We can also use it to create standalone JavaScript.

First we have a script template that is parametrized by a greeting.

```
externalJs :: String -> JStat
externalJs greeting =
    [jmacro|
     window.greet = function (noun)
     {
       alert(`(greeting)` + ' ' + noun);
     }
     |]
```

Notice that we attached the `greet` function to the `window`. The `ToMessage` instance for `JStat` wraps the Javascript in an anonymous function to ensure that statements execute in a local scope. That helps prevents namespace collisions between different external scripts. But, it also means that top-level unhygienic variables will not be global available. So we need to attach them to the `window`.

Next we have a server part with two sub-parts:

```
externalPart :: JMacroPart Response
externalPart = dir "external" $ msum [
```

If `external/script.js` is requested, then we check for a query string parameter `greeting` and generate the script. `toResponse` will automatically convert the script to a `Response` and serve it with the content-type, `text/javascript; charset=UTF-8`:

```
            dir "script.js" $
               do greeting <- optional $ look "greeting"
                  ok $ toResponse $ externalJs (fromMaybe "hello" greeting)
```

Next we have an html page that includes the external script, and calls the `greet` function:

```
         , toResponse <$> defaultTemplate (fromString "external")
            <script type="text/javascript"
                    src="/external/script.js?greeting=Ahoy" />
            <div>
             <h1>Greetings</h1>
             <button onclick=[jmacro| greet('JMacro'); |]>
               Click for a greeting.
             </button>
            </div>
```

```
        ]
```

Instead of attaching the `greet` function to the `window`, we could instead use `jmResponse` to serve the `JStat`. `jmResponse` does not wrap the Javascript in an anonymous function so the `window` work-around is not needed. We do need to use `!` to make sure the name of the `greet2` function is not mangled though:

```
externalJs2 :: String -> JStat
externalJs2 greeting =
    [jmacro|
     function !greet2 (noun)
     {
       alert(`(greeting)` + ' ' + noun);
     }
     |]


externalPart2 :: JMacroPart Response
externalPart2 = dir "external2" $ msum
    [ dir "script.js" $
          do greeting <- optional $ look "greeting"
             jmResponse $ externalJs2 (fromMaybe "hello" greeting)

    , toResponse <$> defaultTemplate (fromString "external 2")
       <script type="text/javascript"
               src="/external2/script.js?greeting=Ahoy" />
       <div>
        <h1>Greetings</h1>
        <button onclick=[jmacro| greet2('JMacro'); |]>
         Click for a greeting.
        </button>
       </div>
    ]
```

Links to demos

Here is a little page that links to all the JMacro demos:

```
demosPart :: JMacroPart Response
demosPart =
    toResponse <$>
      defaultTemplate (fromString "demos") ()
        <ul>
         <li><a href="/hello"    >Hello, JMacro</a></li>
         <li><a href="/attr"     >Hello, Attr</a></li>
```

```
         <li><a href="/endTag"   >Hello, End Tag</a></li>
         <li><a href="/clickMe"  >ClickMe</a></li>
         <li><a href="/clickMe2" >ClickMe2</a></li>
         <li><a href="/functions">Function Names</a></li>
         <li><a href="/fortune"  >Fortune</a></li>
         <li><a href="/weather"  >Weather</a></li>
         <li><a href="/external" >External</a></li>
         <li><a href="/external2" >External 2</a></li>
        </ul>
```

and our routes:

```
handlers :: JMacroPart Response
handlers =
   msum [ dir "hello"     $ helloJMacro
        , dir "attr"      $ helloAttr
        , dir "endTag"    $ helloEndTag
        , dir "clickMe"   $ clickPart
        , dir "clickMe2"  $ clickPart2
        , dir "functions" $ functionNames
        , dir "fortune"   $ fortunePart
        , dir "weather"   $ weatherPart
        , externalPart
        , externalPart2
        , demosPart
        ]
```

Source code for the app is here.

Alternative `IntegerSupply` instance

If you do not like having to use the `StateT` monad transformer to generate names, there are other options. For example, we could use `Data.Unique` to generate unique names:

```
instance IntegerSupply JMacroPart where
 nextInteger =
  fmap (fromIntegral . (`mod` 1024) . hashUnique) (liftIO newUnique)
```

This should be safe as long as you have less than 1024 different JMacro blocks on a single page.

More Information

For more information on using JMacro I recommend reading this wiki page and the tutorial at the top of Language.Javascript.JMacro. The documentation is this tutorial has covered the basics of JMacro, but not everything!

Parsing request data from the QUERY_STRING, cookies, and request body

The `RqData` module is used to extract key/value pairs from the `QUERY_STRING`, cookies, and the request body of a `POST` or `PUT` request.

Hello RqData

Let's start with a simple `hello, world!` example that uses request parameters in the URL.

```
module Main where

import Happstack.Server ( ServerPart, look, nullConf
                        , simpleHTTP, ok)

helloPart :: ServerPart String
helloPart =
    do greeting <- look "greeting"
       noun     <- look "noun"
       ok $ greeting ++ ", " ++ noun

main :: IO ()
main = simpleHTTP nullConf $ helloPart
```

Source code for the app is here.

Now if we visit http://localhost:8000/?greeting=hello&noun=rqdata, we will get the message `hello, rqdata`.

we use the `look` function to look up some keys by name. The `look` function has the type:

```
look :: (Functor m, Monad m, HasRqData m) => String -> m String
```

Since we are using `look` in the `ServerPart` monad it has the simplified type:

```
look :: String -> ServerPart String
```

The `look` function looks up a key and decodes the associated value as a `String`. It assumes the underlying `ByteString` was utf-8 encoded. If you are using some other encoding, then you can use `lookBS` to construct your own lookup function.

If the key is not found, then `look` will fail. In `ServerPart` that means it will call `mzero`.

Handling Submissions

In the previous example we only looked at parameters in the URL. Looking up values from a form submission (a `POST` or `PUT` request) is almost the same. The only difference is we need to first decode the request body using `decodeBody`:

```
{-# LANGUAGE OverloadedStrings #-}
import Control.Monad    (msum)
import Happstack.Server
    ( Response, ServerPart, Method(POST)
    , BodyPolicy(..), decodeBody, defaultBodyPolicy
    , dir, look, nullConf, ok, simpleHTTP
    , toResponse, methodM
    )
import Text.Blaze                           as B
import Text.Blaze.Html4.Strict              as B hiding (map)
import Text.Blaze.Html4.Strict.Attributes as B hiding ( dir, label
                                                      , title)

main :: IO ()
main = simpleHTTP nullConf $ handlers

myPolicy :: BodyPolicy
myPolicy = (defaultBodyPolicy "/tmp/" 0 1000 1000)

handlers :: ServerPart Response
handlers =
    do decodeBody myPolicy
       msum [ dir "hello" $ helloPart
            , helloForm
            ]

helloForm :: ServerPart Response
helloForm = ok $ toResponse $
```

```
    html $ do
      B.head $ do
        title "Hello Form"
      B.body $ do
        form ! enctype "multipart/form-data"
             ! B.method "POST"
             ! action "/hello" $ do
              B.label "greeting: " >> input ! type_ "text"
                                            ! name "greeting"
                                            ! size "10"
              B.label "noun: "      >> input ! type_ "text"
                                            ! name "noun"
                                            ! size "10"
              input ! type_ "submit"
                    ! name "upload"

helloPart :: ServerPart Response
helloPart =
    do methodM POST
       greeting <- look "greeting"
       noun     <- look "noun"
       ok $ toResponse (greeting ++ ", " ++ noun)
```

Source code for the app is here.

Why is `decodeBody` even needed?

The body of the HTTP request is ignored unless we call `decodeBody`. The obvious question is, *"Why isn't the request body automatically decoded?"*

If servers had unlimited RAM, disk, CPU and bandwidth available, then automatically decoding the body would be a great idea. But, since that is generally not the case, we need a way to limit or ignore form submission data that is considered excessive.

A simple solution would be to impose a static quota an all form data submission server-wide. But, in practice, you might want finer granularity of control. By explicitly calling `decodeBody` you can easily configure a site-wide static quota. But you can also easily adapt the quotas depending on the user, particular form, or other criteria.

In this example, we keep things simple and just call `decodeBody` for all incoming requests. If the incoming request is not a `PUT` or `POST` request with `multipart/form-data` then calling `decodeBody` has no side-effects.

Using `BodyPolicy` and `defaultBodyPolicy` to impose quotas

The only argument to `decodeBody` is a `BodyPolicy`. The easiest way to define a `BodyPolicy` is by using the `defaultBodyPolicy` function:

```
defaultBodyPolicy :: FilePath  -- ^ directory to *temporarily*
                               --   store uploaded files in
                  -> Int64     -- ^ max bytes to save to
                               --   disk (files)
                  -> Int64     -- ^ max bytes to hold in RAM
                               --   (normal form values, etc)
                  -> Int64     -- ^ max header size (this only
                               --   affects header in the
                               --   multipart/form-data)
                  -> BodyPolicy
```

In the example, we define this simple policy:

```
myPolicy :: BodyPolicy
myPolicy = (defaultBodyPolicy "/tmp/" 0 1000 1000)
```

Since the form does not do file uploads, we set the file quota to 0. We allow 1000 bytes for the two form fields and 1000 bytes for overhead in the `multipart/form-data` encoding.

Using `decodeBody`

Using `decodeBody` is pretty straight-forward. You simple call it with a `BodyPolicy`. The key things to know are:

1. You must call it anytime you are processing a POST or PUT request and you want to use `look` and friends
2. `decodeBody` only works once per request. The first time you call it the body will be decoded. The second time you call it, nothing will happen, even if you call it with a different policy.

Other tips for using `<form>`

When using the `<form>` element there are two important recommendations you should follow:

1. Set the `enctype` to `multipart/form-data`. This is especially important for forms which contain file uploads.
2. Make sure to set `method` to `POST` or the form values will show up in the URL as query parameters.

File Uploads

The `lookFile` function is used to extract an uploaded file:

```
lookFile :: String -> RqData (FilePath, FilePath, ContentType)
```

It returns three values:

1. The location of the temporary file which holds the contents of the file
2. The *local* filename supplied by the browser. This is typically the name of the file on the users system.
3. The `content-type` of the file (as supplied by the browser)

The temporary file will be automatically deleted after the `Response` is sent. Therefore, it is essential that you move the file from the temporary location.

In order for file uploads to work correctly, it is also essential that your `<form>` element contains the attributes `enctype="multipart/form-data"` and `method="POST"`

The following example has a form which allows a user to upload a file. We then show the temporary file name, the uploaded file name, and the content-type of the file. In a real application, the code should use `System.Directory.renameFile` (or similar) to move the temporary file to a permanent location. This example looks a bit long, but most of the code is just HTML generation using BlazeHtml. The only really new part is the use of the `lookFile` function. Everything else should already have been covered in previous sections. So if you don't understand something, try looking in earlier material.

```
{-# LANGUAGE OverloadedStrings #-}
import Control.Monad    (msum)
import Happstack.Server
    ( Response, ServerPart, Method(GET, POST), defaultBodyPolicy
    , decodeBody, dir, lookFile, method, nullConf, ok
    , simpleHTTP, toResponse )
import           Text.Blaze                          ((!))
import qualified Text.Blaze                          as H
import qualified Text.Blaze.Html4.Strict             as H
import qualified Text.Blaze.Html4.Strict.Attributes as A

main :: IO ()
main = simpleHTTP nullConf $ upload

upload :: ServerPart Response
upload =
    do decodeBody (defaultBodyPolicy "/tmp/" (10*10^6) 1000 1000)
       msum [ dir "post" $ post
            , uploadForm
            ]
```

```
uploadForm :: ServerPart Response
uploadForm =
    do method GET
       ok $ toResponse $
        H.html $ do
         H.head $ do
         H.title "Upload Form"
         H.body $ do
          H.form ! A.enctype "multipart/form-data"
                 ! A.method "POST"
                 ! A.action "/post" $ do
                   H.input ! A.type_ "file" ! A.name "file_upload" ! A.size "40"
                   H.input ! A.type_ "submit" ! A.value "upload"

post :: ServerPart Response
post =
   do method POST
      r <- lookFile "file_upload"
      -- renameFile (tmpFile r) permanentName
      ok $ toResponse $
         H.html $ do
           H.head $ do
             H.title "Post Data"
           H.body $ mkBody r
    where
      mkBody (tmpFile, uploadName, contentType) = do
        H.p (H.toHtml $ "temporary file: " ++ tmpFile)
        H.p (H.toHtml $ "uploaded name:  " ++ uploadName)
        H.p (H.toHtml $ "content-type:   " ++ show contentType)
```

Source code for the app is here.

File uploads important reminder

Remember that you must move the temporary file to a new location or it will be garbage collected after the 'Response' is sent. In the example code we do *not* move the file, so it is automatically deleted.

Limiting lookup to `QUERY_STRING` or request body

By default, `look` and friends will search both the QUERY_STRING the request body (aka, `POST`/`PUT` data) for a key. But sometimes we want to specify that only the `QUERY_STRING` or request body should be searched. This can be done by using the `body` and `queryString` filters:

```
body        :: (HasRqData m) => m a -> m a
queryString :: (HasRqData m) => m a -> m a
```

Using these filters we can modify `helloPart` so that the `greeting` must come from the QUERY_STRING and the `noun` must come from the request body:

```
helloPart :: ServerPart String
helloPart =
    do greeting <- queryString $ look "greeting"
       noun     <- body        $ look "noun"
       ok $ greeting ++ ", " ++ noun
```

`queryString` and `body` act as filters which only pass a certain subset of the data through. If you were to write:

```
greetingRq :: ServerPart String
greetingRq =
    body (queryString $ look "greeting")
```

This code would never match anything because the `body` filter would hide all the `QUERY_STRING` values, and the `queryString` filter would hide all the request body values, and hence, there would be nothing left to search.

Using the RqData for better error reporting

So far we have been using the `look` function in the `ServerPart` monad. This means that if any `look` fails, that handler fails. Unfortunately, we are not told what parameter was missing – which can be very frustrating when you are debugging your code. It can be even more annoying if you are providing a web service, and whenever a developer forgets a parameter, they get a 404 with no information about what went wrong.

So, if we want better error reporting, we can use functions like `look` in the `RqData Applicative Functor`.

We can use `getDataFn` to run the `RqData`:

```
getDataFn :: (HasRqData m, ServerMonad m, MonadIO m) =>
             RqData a
          -> m (Either [String] a)
```

```
module Main where

import Control.Applicative  ((<$>), (<*>))
import Happstack.Server     ( ServerPart, badRequest, nullConf
                            , ok, simpleHTTP)
import Happstack.Server.RqData (RqData, look, getDataFn)

helloRq :: RqData (String, String)
```

```
helloRq =
    (,) <$> look "greeting" <*> look "noun"

helloPart :: ServerPart String
helloPart =
    do r <- getDataFn helloRq
       case r of
         (Left e) ->
             badRequest $ unlines e
         (Right (greet, noun)) ->
             ok $ greet ++ ", " ++ noun

main :: IO ()
main = simpleHTTP nullConf $ helloPart
```

Source code for the app is here.

If we visit http://localhost:8000/?greeting=hello&noun=world, we will get our familiar greeting `hello, world`. But if we leave off the query parameters http://localhost:8000/, we will get a list of errors:

```
Parameter not found: greeting
Parameter not found: noun
```

We could use the `Monad` instance `RqData` to build the request. However, the monadic version will only show us the *first* error that is encountered. So would have only seen that the `greeting` was missing. Then when we added a `greeting` we would have gotten a new error message saying that `noun` was missing.

In general, improved error messages are not going to help people visiting your website. If the parameters are missing it is because a form or link they followed is invalid. There are two places where there error messages are useful:

1. When you are developing and debugging your site
2. Reporting errors to users of your web service API

If you are providing a REST API for developers to use, they are going to be a lot happier if they get a detailed error messages instead of a plain old 404.

Using `checkRq`

Sometimes the representation of a value as a request parameter will be different from the representation required by `Read`. We can use `checkRq` to lift a custom parsing function into `RqData`.

```
checkRq :: (Monad m, HasRqData m) => m a -> (a -> Either String b) -> m b
```

In this example we create a type `Vote` with a custom parsing function:

```
module Main where

import Control.Applicative ((<$>), (<*>))
import Happstack.Server
    ( ServerPart, badRequest
    , nullConf, ok, simpleHTTP)
import Happstack.Server.RqData
    ( RqData, checkRq
    , getDataFn, look, lookRead)

data Vote = Yay | Nay
    deriving (Eq, Ord, Read, Show, Enum, Bounded)

parseVote :: String -> Either String Vote
parseVote "yay" = Right Yay
parseVote "nay" = Right Nay
parseVote str   =
    Left $ "Expecting 'yay' or 'nay' but got: " ++ str

votePart :: ServerPart String
votePart =
    do r <- getDataFn (look "vote" `checkRq` parseVote)
       case r of
         (Left e) ->
             badRequest $ unlines e
         (Right i) ->
             ok $ "You voted: " ++ show i

main :: IO ()
main = simpleHTTP nullConf $ votePart
```

Source code for the app is here.

Now if we visit http://localhost:8000/?vote=yay, we will get the message:

```
You voted: Yay
```

If we visit http://localhost:8000/?vote=yes, we will get the error:

```
Expecting 'yay' or 'nay' but got: yes
```

Other uses of `checkRq`

Looking again at the type for `checkRq` we see that function argument is fairly general – it is not restricted to just string input:

```
checkRq :: RqData a -> (a -> Either String b) -> RqData b
```

So, `checkRq` is not limited to just parsing a `String` into a value. We could use it, for example, to validate an existing value. In the following example we use `lookRead "i"` to convert the value `i` to an `Int`, and then we use `checkRq` to ensure that the value is within range:

```
module Main where

import Control.Applicative ((<$>), (<*>))
import Happstack.Server
    (ServerPart, badRequest, nullConf, ok, simpleHTTP)
import Happstack.Server.RqData
    (RqData, checkRq, getDataFn, look, lookRead)

inRange :: (Show a, Ord a) => a -> a -> a -> Either String a
inRange lower upper a
    | lower <= a && a <= upper = Right a
    | otherwise =
        Left (show a ++ " is not between " ++
              show lower ++ " and " ++ show upper)

oneToTenPart :: ServerPart String
oneToTenPart = do
  r <- getDataFn (lookRead "i" `checkRq`(inRange (1 :: Int) 10))
  case r of
    (Left e) ->
        badRequest $ unlines e
    (Right i) ->
        ok $ "You picked: " ++ show i

main :: IO ()
main = simpleHTTP nullConf $ oneToTenPart
```

Source code for the app is here.

Now if we visit http://localhost:8000/?i=10, we will get the message:

```
 $ curl http://localhost:8000/?i=10
You picked: 10
```

But if we pick an out of range value http://localhost:8000/?i=113, we will get the message:

```
 $ curl http://localhost:8000/?i=113
113 is not between 1 and 10
```

Looking up optional parameters

Sometimes query parameters are optional. You may have noticed that the `RqData` module does not seem to provide any functions for dealing with optional values. That is because we can just use the `Alternative` class from `Control.Applicative` which provides the function `optional` for us:

```
optional :: Alternative f => f a -> f (Maybe a)
```

Here is a simple example where the `greeting` parameter is optional:

```
module Main where

import Control.Applicative ((<$>), (<*>), optional)
import Happstack.Server
    (ServerPart, look, nullConf, ok, simpleHTTP)

helloPart :: ServerPart String
helloPart =
    do greet <- optional $ look "greeting"
       ok $ (show greet)

main :: IO ()
main = simpleHTTP nullConf $ helloPart
```

Source code for the app is here.

If we visit http://localhost:8000/?greeting=hello, we will get `Just "hello"`.

if we leave off the query parameters we get http://localhost:8000/, we will get `Nothing`.

Working with Cookies

HTTP is a stateless protocol. Each incoming `Request` is processed with out any memory of any previous communication with the client. Though, from using the web, you know that it certainly doesn't feel that way. A website can remember that you logged in, items in your shopping cart, etc. That functionality is implemented by using `Cookies`.

When the server sends a `Response` to the client, it can include a special `Response` header named `Set-Cookie`, which tells the client to remember a certain `Cookie`. A `Cookie` has a name, a string value, and some extra control data, such as a lifetime for the cookie.

The next time the client talks to the server, it will include a copy of the `Cookie` value in its `Request` headers. One possible use of cookies is to store a session id. When the client submits the cookie, the server can use the session id to look up information about the client and remember who they are. Sessions and session

ids are not built-in to the HTTP specification. They are merely a common idiom which is provided by many web frameworks.

Simple Cookie Demo

The cookie interface is pretty small. There are two parts to the interface: setting a cookie and looking up a cookie.

To create a `Cookie` value, we use the `mkCookie` function:

```
-- | create a 'Cookie'
mkCookie  :: String -- ^ cookie name
          -> String -- ^ cookie value
          -> Cookie
```

Then we use the `addCookie` function to send the cookie to the user. This adds the `Set-Cookie` header to the `Response`. So the cookie will not actually be set until the `Response` is sent.

```
-- | add the 'Cookie' to the current 'Response'
addCookie :: (MonadIO m, FilterMonad Response m) =>
             CookieLife
          -> Cookie
          -> m ()
```

The first argument of `addCookie` specifies how long the browser should keep the cookie around. See the cookie lifetime section for more information on `CookieLife`.

To lookup a cookie, we use some `HasRqData` functions. There are only three cookie related functions:

```
-- | lookup a 'Cookie'
lookCookie :: (Monad m, HasRqData m) =>
              String -- ^ cookie name
           -> m Cookie

-- | lookup a 'Cookie' and return its value
lookCookieValue :: (Functor m, Monad m, HasRqData m) =>
                   String -- ^ cookie name
                -> m String

-- | look up a 'Cookie' value and try to convert it using 'read'
readCookieValue :: (Functor m, Monad m, HasRqData m, Read a) =>
                   String -- ^ cookie name
                -> m a
```

The cookie functions work just like the other `HasRqData` functions. That means you can use `checkRq`, etc.

The following example puts all the pieces together. It uses the cookie to store a simple counter specifying how many requests have been made:

```
module Main where

import Control.Monad.Trans ( liftIO )
import Control.Monad       ( msum, mzero )
import Happstack.Server
    ( CookieLife(Session), Request(rqPaths), ServerPart
    , addCookie , askRq, look, mkCookie, nullConf
    , ok, readCookieValue, simpleHTTP )

homePage :: ServerPart String
homePage = msum
  [ do rq <- askRq
       liftIO $ print (rqPaths rq)
       mzero
  , do requests <- readCookieValue "requests"
       addCookie Session (mkCookie "requests"
                          (show (requests + (1 :: Int))))
       ok $ "You have made " ++ show requests ++
            " requests to this site."
  , do addCookie Session (mkCookie "requests" (show 2))
       ok $ "This is your first request to this site."
  ]

main :: IO ()
main = simpleHTTP nullConf $ homePage
```

Source code for the app is here.

Now if you visit http://localhost:8000/ you will get a message like:

```
This is your first request to this site.
```

If you hit reload you will get:

```
You have made 3 requests to this site.
```

Now wait a second! How did we go from 1 to 3, what happened to 2? The browser will send the cookie with every request it makes to the server. In this example, we ignore the request path and send a standard response to every request that is made. The browser first requests the page, but it also requests the `favicon.ico` for the site. So, we are really getting two requests everytime we load the page. Hence the counting by twos. It is important to note that the browser does not just send the cookie when it is expecting an html page – it will send it when it is expecting a jpeg, a css file, a js, or anything else.

There is also a race-condition bug in this example. See the cookie issues section for more information.

Cookie Lifetime

When you set a cookie, you also specify the lifetime of that cookie. Cookies are referred to as `session cookies` or `permanent cookies` depending on how their lifetime is set.

session cookie A cookie which expires when the browser is closed.

permanent cookie A cookie which is saved (to disk) and is available even if the browser is restarted. The expiration time is set by the server.

The lifetime of a `Cookie` is specified using the `CookieLife` type:

```
-- | the lifetime of the cookie
data CookieLife
  = Session          -- ^ expire when the browser is closed
  | MaxAge Seconds   -- ^ expire after the specified
                     --   number of seconds
  | Expires UTCTime  -- ^ expire at a specific date and time
  | Expired          -- ^ expire immediately
```

If you are intimately familiar with cookies, you may know that cookies have both an `expires` directive and a `max-age` directive, and wonder how they related to the constructors in `CookieLife`. Internet Explorer only supports the obsolete `expires` directive, instead of newer `max-age` directive. Most other browser will honor the `max-age` directive over `expires` if both are present. To make everyone happy, we always set both.

So, when setting `CookieLife` you can use `MaxAge` or `Expires` – which ever is easiest, and the other directive will be calculated automatically.

Deleting a Cookie

There is no explicit `Response` header to delete a cookie you have already sent to the client. But, you can convince the client to delete a cookie by sending a new version of the cookie with an expiration date that as already come and gone. You can do that by using the `Expired` constructor. Or, you can use the more convenient, `expireCookie` function.

```
-- | Expire the cookie immediately and set the cookie value to ""
expireCookie :: (MonadIO m, FilterMonad Response m) =>
                String  -- ^ cookie name
             -> m ()
```

Cookie Issues

Despite their apparently simplicity, `Cookies` are the source of many bugs and security issues in web applications. Here are just a few of the things you need to

keep in mind.

Security issues

To get an understanding of cookie security issues you should search for:

- cookie security issues
- cookie XSS

One important thing to remember is that the user can modify the cookie. So it would be a bad idea to do, `addCookie Session (mkCookie "userId" "1234")` because the user could modify the cookie and change the userId at will to access other people's accounts.

Also, if you are not using `https` the cookie will be sent unencrypted.

Delayed Effect

When you call `addCookie` the `Cookie` will not be available until after that `Response` has been sent and a new `Request` has been received. So the following code will not work:

```
do addCookie Session (mkCookie "newCookie" "newCookieValue")
   v <- look "newCookie"
   ...
```

The first time it runs, `look` will fail because the cookie was not set in the current `Request`. Subsequent times `look` will return the old cookie value, not the new value.

Cookie Size

Browsers impose limits on how many cookies each site can issue, and how big those cookies can be. The RFC recommends browsers accept a minimum of 20 cookies per site, and that cookies can be at least 4096 bytes in size. But, implementations may vary. Additionally, the cookies will be sent with every request to the domain. If your page has dozens of images, the cookies will be sent with every request. That can add a lot of overhead and slow down site loading times.

A common alternative is to store a small session id in the cookie, and store the remaining information on the server, indexed by the session id. Though that brings about its own set of issues.

One way to avoid having cookies sent with every image request is to host the images on a different sub-domain. You might issues the cookies to www.example.org, but host images from images.example.org. Note that you do not actually have to run two servers in order to do that. Both domains can point to the same IP address and be handled by the same application. The app itself may not even distinguish if the requests were sent to `images` or `www`.

Server Clock Time

In order to calculate the `expires` date from the `max-age` or the `max-age` from the `expires` date, the server uses `getCurrentTime`. This means your system clock should be reasonably accurate. If your server is not synchronized using `NTP` or something similar it should be.

Cookie Updates are Not Atomic

Cookie updates are not performed in any sort of atomic manner. As a result, the simple cookie demo contains a race condition. We get the `Cookie` value that was included in the `Request` and use it to create an updated `Cookie` value in the `Response`. But remember that the server can be processing many requests in parallel and the browser can make multiple requests in parallel. If the browser, for example, requested 10 images at once, they would all have the same initial cookie value. So, even though they all updated the counter by 1, they all started from the same value and ended with the same value. The count could even go backwards depending on the order `Requests` are received and `Responses` are processed.

Other Cookie Features

The `mkCookie` function uses some default values for the `Cookie`. The `Cookie` type itself includes extra parameters you might want to control such as the cookie path, the secure cookie option, etc.

Serving Files from Disk

Happstack can be used to serve static files from disk, such as `.html`, `.jpg`, etc.

The file serving capabilities can be divided into two categories:

1. Serving files from a directory based on a direct mapping of a portion of the URI to file names on the disk
2. Serving an specific, individual file on disk, whose name may be different from the URI

Serving Files from a Directory

The most common way to serve files is by using `serveDirectory`:

```
data Browsing  = EnableBrowsing | DisableBrowsing

serveDirectory :: ( WebMonad Response m, ServerMonad m, FilterMonad Response m
                  , MonadIO m, MonadPlus m
                  ) =>
                  Browsing    -- ^ enable/disable directory browsing
                -> [FilePath]  -- ^ index file names
                -> FilePath    -- ^ file/directory to serve
                -> m Response
```

For example:

```
serveDirectory EnableBrowsing ["index.html"] "path/to/directory/on/disk"
```

If the requested path does not map to a file or directory, then `serveDirectory` returns `mzero`.

If the requested path is a file then the file is served normally using `serveFile`.

When a directory is requested, `serveDirectory` will first try to find one of the index files (in the order they are listed). If that fails, it will show a directory listing if `EnableBrowsing`, otherwise it will return `forbidden "Directory index forbidden"`.

The formula for mapping the URL to a file on disk is just what you would expect:

```
path/to/directory/on/disk </> unconsumed/portion/of/request/url
```

So if the handler is:

```
 dir "static" $
  serveDirectory EnableBrowsing ["index.html"] "/srv/mysite/data"
```

And the request URL is:

```
http://localhost/static/foo/bar.html
```

Then we are going to look for:

```
/srv/mysite/data </> foo/bar.html => /srv/mysite/data/foo/bar.html
```

The following demo will allow you to browse the directory that the server is running in. (So be careful where you run it).

```
module Main where

import Happstack.Server ( Browsing(EnableBrowsing), nullConf
                        , serveDirectory, simpleHTTP
                        )

main :: IO ()
main = simpleHTTP nullConf $ serveDirectory EnableBrowsing [] "."
```

Source code for the app is here.

Simply run it and point your browser at http://localhost:8000/

File Serving Security

The request URL is sanitized so that users can not escape the top-level directory by adding extra `..` or `/` characters to the URL.

The file serving code *will* follow symlinks. If you do not want that behavior then you will need to roll your own serving function. See the section on *Advanced File Serving* for more information.

Serving a Single File

Sometimes we want to serve files from disk whose name is not a direct mapping from the URL. For example, let's say that you have an image and you want to allow the client to request the images in different sizes by setting a query parameter. e.g.

```
http://localhost:8000/images/photo.jpg?size=medium
```

Clearly, we can not just map the path info portion of the URL to a file disk, because all the different sizes have the same name – only the query parameter is different. Instead, the application will use some custom algorithm to calculate where the image lives on the disk. It may even need to generate the resized image on-demand. Once the application knows where the file lives on disk it can use `serveFile` to send that file as a `Response` using `sendFile`:

```
serveFile :: ( ServerMonad m
             , FilterMonad Response m
             , MonadIO m
             , MonadPlus m
             ) =>
             (FilePath -> m String) -- ^ function for determining
                                    --   content-type of file.
                                    --   Usually 'asContentType'
                                    --   or 'guessContentTypeM'
          -> FilePath               -- ^ path to the file to serve
          -> m Response
```

The first argument is a function which calculates the mime-type for a `FilePath`. The second argument is path to the file to send. So we might do something like:

```
serveFile (guessContentTypeM mimeTypes) "/srv/photos/photo.jpg"
```

Note that even though the file is named `photo_medium.jpg` on the disk, that name is not exposed to the client. They will only see the name they requested, i.e., `photo.jpg`.

`guessContentTypeM` will guess the content-type of the file by looking at the filename extension. But, if our photo app only supports JPEG files, there is no need to guess. Furthermore, the name of the file on the disk may not even have the proper extension. It could just be the md5sum of the file or something. So we can also hardcode the correct content-type:

```
serveFile (asContentType "image/jpeg") "/srv/photos/photo.jpg"
```

The following, example attempts to serve its own source code for any incoming request.

```
module Main where

import Happstack.Server ( asContentType, nullConf
                        , serveFile, simpleHTTP)

main :: IO ()
main =
 simpleHTTP nullConf $
  serveFile (asContentType "text/x-haskell") "FileServingSingle.hs"
```

Source code for the app is here.

Advanced File Serving

`serveDirectory` and `serveFile` should cover a majority of your file serving needs. But if you want something a little different, it is also possible to roll-your-own solution. The `Happstack.Server.FileServe.BuildingBlocks` module contains all the pieces used to assemble the high-level `serveDirectory` and `serveFile` functions. You can reuse those pieces to build your own custom serving functions. For example, you might want to use a different method for calculating the mime-types, or perhaps you want to create a different look-and-feel for directory browsing, or maybe you want to use something other than `sendFile` for sending the files. I recommend starting by copying the source for `serveDirectory` or `serveFile` and then modifying it to suit your needs.

Type-Safe `Form` processing using `reform`

`reform` is a library for creating type-safe, composable, and validated HTML forms. It is built around applicative functors and is based on the same principles as `formlets` and `digestive-functors <= 0.2`.

The core `reform` library is designed to be portable and can be used with a wide variety of Haskell web frameworks and template solutions – though only a few options are supported at the moment.

The most basic method of creating and processing forms with out the assistance of `reform` is to:

1. create a `<form>` tag with the desired elements by hand
2. write code which processes the form data set and tries to extract a value from it

The developer will encounter a number of difficulties using this method:

1. the developer must be careful to use the same `name` field in the HTML and the code.
2. if a new field is added to the form, the code must be manually updated. Failure to do so will result in the new field being silently ignored.
3. form fragments can not be easily combined because the `name` or `id` fields might collide. Additionally, there is no simple way to combine the validation/value extraction code.
4. if the form fails to validate, it is difficult to redisplay the form with the error messages and data that was submitted.

`reform` solves these problems by combining the view generation code and validation code into a single `Form` element. The `Form` elements can be safely combined to create more complex forms.

In theory, `reform` could be applied to other domains, such as command-line or GUI applications. However, `reform` is based around the pattern of:

1. generate the entire form at once
2. wait until the user has filled out all the fields and submitted it
3. process the results and generate an answer or redisplay the form with validation errors

For most interactive applications, there is no reason to wait until the entire form has been filled out to perform validation.

Brief History

`reform` is an extension of the OCaml-based `formlets` concept originally developed by Ezra Cooper, Sam Lindley, Philip Wadler and Jeremy Yallop. The original `formlets` code was ported to Haskell as the `formlets` library, and then revamped again as the `digestive-functors <= 0.2` library.

`digestive-functors` 0.3 represents a major break from the traditional `formlets` model. The primary motivation behind `digestive-functors` 0.3 was (mostly likely) to allow the separation of validators from the view code. This allows library authors to define validation for forms, while allowing the library users to create the view for the forms. It also provides a mechanism to support templating systems like `Heist`, where the view is defined in an external XML file rather than Haskell code.

In order to achieve this, `digestive-functors` 0.3 unlinks the validation and view code and requires the developers to stitch them back together using `String` based names. This, of course, leads to runtime errors. If the library author adds new required fields to the validator, the user gets no compile time warnings or errors to let them know their code is broken.

The `Reform` library is a heavily modified fork of `digestive-functors` 0.2. It builds on the the traditional `formlets` safety and style and extends it to allow view and validation separation in a type-safe manner.

You can find the original papers on `formlets` here.

Hello `Form`!

You will need to install the following optional packages for this section:

```
cabal install reform reform-happstack reform-hsp
```

The easiest way to learn `Reform` is through example. We will start with a simple form that does not require any special validation. We will then extend the form, adding some simple validators. And then we will show how we can split the validation and view for our form into separate libraries.

This example uses Happstack for the web server and HSP for the templating library.

First we have some pragmas:

```
{-# LANGUAGE FlexibleContexts, FlexibleInstances,
    MultiParamTypeClasses, ScopedTypeVariables,
    TypeFamilies, TypeSynonymInstances,
    QuasiQuotes, OverloadedStrings #-}
module Main where
```

And then some imports. We import modules from three different `reform` packages: the core `reform` library, the `reform-happstack` package, and the `reform-hsp` package:

```
import Control.Applicative
import Control.Applicative.Indexed
    (IndexedFunctor(..), IndexedApplicative(..))
import Control.Monad                (msum)
import Data.Text.Lazy               (Text)
import qualified Data.Text.Lazy     as Lazy
import qualified Data.Text          as Strict
import Happstack.Server
import Happstack.Server.XMLGenT     ()
import Happstack.Server.HSP.HTML    ()
import HSP
import HSP.Monad                    (HSPT(..))
import Language.Haskell.HSX.QQ      (hsx)
import Text.Reform
    ( CommonFormError(..), Form, FormError(..), Proof(..), (++>)
    , (<++), commonFormErrorStr, decimal, prove
    , transformEither, transform )
import Text.Reform.Happstack
import Text.Reform.HSP.Text
```

Next we will create a type alias for our application's server monad:

```
type AppT m  = XMLGenT (HSPT XML (ServerPartT m))
type AppT' m = HSPT XML (ServerPartT m)
```

We will also want a function that generates a page template for our app:

```
appTemplate :: ( Functor m, Monad m
               , EmbedAsChild (AppT' m) headers
               , EmbedAsChild (AppT' m) body
               ) =>
               Text    -- ^ contents of <title> tag
            -> headers -- ^ extra content for <head> tag.
                       --   use () for nothing
            -> body    -- ^ contents of <body> tag
            -> AppT m Response
appTemplate title headers body =
```

```
  toResponse <$> [hsx|
    <html>
     <head>
      <title><% title %></title>
      <% headers %>
     </head>
     <body>
      <% body %>
     </body>
    </html>
    |]
```

Forms have the type `Form` which looks like:

```
newtype Form m input error view proof a = Form { ... }
```

As you will note it is heavily parameterized:

`m` a monad which can be used to validate the result `input`
: the framework specific type containing the fields from the form data set.
`error`
: An application specific type for form validation errors. `view`
: The type of the view for the form. `proof`
: A datatype which names something that has been proven about the result.
`a`
: The value returned when the form data set is successfully decoded and validated.

In order to keep our type signatures sane, it is convenient to create an application specific type alias for the `Form` type:

```
type SimpleForm =
 Form (AppT IO) [Input] AppError [AppT IO XML] ()
```

`AppError` is an application specific type used to report form validation errors:

```
data AppError
    = Required
    | NotANatural String
    | AppCFE (CommonFormError [Input])
      deriving Show
```

Instead of having one error type for all the forms, we could have per-form error types – or even just use `String`. The advantage of using a type is that it makes it easier to provide I18N translations, or for users of a library to customize the text of the error messages. The disadvantage of using a custom type over a plain `String` is that it can make it more difficult to combine forms into larger forms since they must all have the same error type. Additionally, it is a bit more work to create the error type and the `FormError` instance.

We will want an `EmbedAsChild` instance so that we can easily embed the errors in our HTML:

```
instance (Functor m, Monad m) =>
    EmbedAsChild (AppT' m) AppError where
  asChild Required          =
    asChild $ "required"

  asChild (NotANatural str) =
    asChild $ "Could not decode as a positive integer: " ++ str

  asChild (AppCFE cfe)      =
     asChild $ commonFormErrorStr show cfe

instance (Functor m, Monad m) =>
         EmbedAsChild (AppT' m) Strict.Text where
    asChild t = asChild (Lazy.fromStrict t)

instance (Functor m, Monad m) =>
         EmbedAsAttr (AppT' m) (Attr Text Strict.Text) where
    asAttr (n := v) = asAttr (n := Lazy.fromStrict v)
```

The error type also needs a `FormError` instance:

```
instance FormError AppError where
    type ErrorInputType AppError = [Input]
    commonFormError = AppCFE
```

Internally, `reform` has an error type `CommonFormError` which is used to report things like missing fields and other internal errors. The `FormError` class is used to lift those errors into our custom error type.

Now we have the groundwork laid to create a simple form. Let's create a form that allows users to post a message. First we will want a type to represent the message – a simple record will do:

```
data Message = Message
    { name    :: Strict.Text -- ^ the author's name
    , title   :: Strict.Text -- ^ the message title
    , message :: Strict.Text -- ^ contents of the message
    } deriving (Eq, Ord, Read, Show)
```

and a simple function to render the `Message` as `XML`:

```
renderMessage :: ( Functor m
                 , Monad m
                 , EmbedAsChild (AppT' m) Strict.Text) =>
                 Message -> AppT m XML
renderMessage msg =
   [hsx|
```

```
    <dl>
      <dt>name:</dt>    <dd><% name msg    %></dd>
      <dt>title:</dt>   <dd><% title msg   %></dd>
      <dt>message:</dt> <dd><% message msg %></dd>
    </dl>
   |]
```

Now we can create a very basic form:

```
postForm :: SimpleForm Message
postForm =
  Message
   <$> labelText "name:"             ++> inputText ""      <++ br
   <*> labelText "title: "           ++> inputText ""      <++ br
   <*> (labelText "message:" <++ br) ++> textarea 80 40 "" <++ br
   <*  inputSubmit "post"
```

This form contains all the information needed to generate the form elements and to parse the submitted form data set and extract a `Message` value.

The following functions come from `reform-hsp`. `reform-blaze` provides similar functions.

- `label` function creates a `<label>` element using the supplied label.
- `inputText` function creates a `<input type="text">` input element using the argument as the initial value.
- `inputSubmit` function creates a `<input type="submit">` using the argument as the value.
- `textarea` function creates `<textearea>`. The arguments are the number of cols, rows, and initial contents.
- `br` functions creates a `Form` element that doesn't do anything except insert a `<br>` tag.

The `<$>`, `<*>` and `<*` operators come from `Control.Applicative`. If you are not familiar with applicative functors then you will want to read a tutorial such as this one.

`++>` comes from the `reform` library and has the type:

```
(++>) :: (Monad m, Monoid view) =>
         Form m input error view () ()
      -> Form m input error view proof a
      -> Form m input error view proof a
```

The `++>` operator is similar to the `*>` operator with one important difference. If we were to write:

```
label "name: " *> inputText
```

then the `label` and `inputText` would each have unique `FormId` values. But when we write:

```
label "name: " ++> inputText
```

they have the same `FormId` value. The `FormId` value is typically used to create unique `name` and `id` attributes for the form elements. But, in the case of `label`, we want the `for` attribute to refer to the `id` of the element it is labeling. There is also a similar operator <++ for when you want the label after the element.

We also use `<++` and `++>` to attach error messages to form elements.

Using the `Form`

The easiest way to use `Form` is with the `happstackEitherForm` function:

```
postPage :: AppT IO Response
postPage =
  dir "post" $ do
   let action = "/post" :: Text
   result <- happstackEitherForm (form action) "post" postForm
   case result of
     (Left formHtml) ->
         appTemplate "post" () formHtml
     (Right msg)     ->
         appTemplate "Your Message" () $ renderMessage msg
```

`happstackEitherForm` has the type:

```
happstackEitherForm :: (Happstack m) =>
     ([(Text, Text)] -> view -> view)  -- ^ wrap raw form html
                                       --   inside a <form> tag
  -> Text                              -- ^ form prefix
  -> Form m [Input] error view proof a -- ^ Form to run
  -> m (Either view a)                 -- ^ Result
```

For a `GET` request, `happstackEitherForm` will view the form with `NoEnvironment`. It will always return `Left view`.

For a `POST` request, `happstackEitherForm` will attempt to validate the form using the form submission data. If successful, it will return `Right a`. If unsuccessful, it will return `Left view`. In this case, the view will include the previously submitted data plus any error messages.

Note that since `happstackEitherForm` is intended to handle both `GET` and `POST` requests, it is important that you do not have any `method` calls guarding `happstackEitherForm` that would interfere.

The first argument to `happstackEitherForm` is a function what wraps the view inside a `<form>` element. This function will typically be provided by template

specific reform package. For example, `reform-hsp` exports:

```
-- | create <form action=action
--                method="POST"
--                enctype="multipart/form-data">
form :: (XMLGenerator x, EmbedAsAttr x (Attr Text action)) =>
        action                    -- ^ action url
     -> [(Text,Text)]             -- ^ extra hidden fields
     -> [XMLGenT x (XMLType x)] -- ^ children
     -> [XMLGenT x (XMLType x)]
```

The first argument to `form` is the attribute to use for the `action` attribute. The other arguments will be filled out by `happstackEitherForm`.

The second argument to `happstackEitherForm` is a unique `String`. This is used to ensure that each `<form>` on a page generates unique `FormId` values. This is required since the `FormId` is typically used to generate `id` attributes, which must be unique.

The third argument to `happstackEitherForm` is the the form we want to use.

`reform` function

`happstackEitherForm` is fairly straight-forward, but can be a bit tedious at times:

1. having to do `case result of` is a bit tedious.
2. when using `HSP`, it is a bit annoying that the `happstackEitherForm` appears outside of the rest of the page template

These problems are even more annoying when a page contains multiple forms.

`reform-happstack` exports `reform` which can be used to embed a `Form` directly inside an `HSP` template:

```
postPage2 :: AppT IO Response
postPage2 =
 dir "post2" $
  let action = ("/post2" :: Text) in
  appTemplate "post 2" () $[hsx|
   <% reform (form action) "post2" displayMsg Nothing postForm %>
  |]
 where
  displayMsg msg =
    appTemplate "Your Message" () $ renderMessage msg
```

`reform` has a pretty intense looking type signature but it is actually pretty straight-forward, and similar to `eitherHappstackForm`:

```
reform :: ( ToMessage b
          , Happstack m
          , Alternative m
          , Monoid view) =>
    ([(Text, Text)] -> view -> view) -- ^ wrap raw form html inside
                                     -- a @\<form\>@ tag
 -> Text                             -- ^ prefix
 -> (a -> m b)                       -- ^ success handler used when
                                     --   form validates
 -> Maybe ([(FormRange, error)] -> view -> m b) -- ^ failure handler
                                                --   used when form
                                                --   does not validate
 -> Form m [Input] error view proof a           -- ^ the formlet
 -> m view
reform toForm prefix success failure form = ...
```

`toForm` should wrap the view returned by the form in a `<form>` tag. Here we use the `form` function from `reform-happstack`. The first argument to `form` is the `action` url. `prefix` the `FormId` prefix to use when rendering this form. `handleSuccess` is the function to call if the form validates successfully. It gets the value extracted from the form. `hHandleFailure` is a function to call if for validation fails. If you pass in `Nothing` then the form will simple by redisplayed in the original context. `form` is the `Form` to process.

Cross-Site Request Forgery (CSRF) Protection

The `happstackEitherForm` and `reform` functions also have a hidden benefit – they provide cross-site request forgery (CSRF) protection, using the double-submit method. When the `<form>` is generated, the `reform` or `happstackEitherForm` function will create a secret token and add it to a hidden field in the form. It will also put the secret token in a cookie. When the user submits the form, the `reform` function will check that the value in the cookie and the hidden field match. This prevents rogue sites from tricking users into submitting forms, because the rogue site can not get access to the secret token in the user's cookie.

That said, if your site is vulnerable to cross site script (XSS) attacks, then it may be possible for a remote site to steal the cookie value.

Benefits So Far

The form we have so far is very simple. It accepts any input, not caring if the fields are empty or not. It also does not try to convert the `String` values to

another type before adding them to the record.

However, we do still see a number of benefits. We specified the form once, and from that we automatically extract the code to generate HTML and the code to extract the values from the form data set. This adheres to the DRY (don't repeat yourself) principle. We did not have to explicitly name our fields, keep the names in-sync in two different places, worry if the HTML and processing code contain the same set of fields, or worry if a name/id has already been used. Additionally, we get automatic CSRF protection.

`Form` with Simple Validation

The next step is to perform some validation on the input fields. If the fields validate successfully, then we get a `Message`. But if the input fails to validate, then we will automatically regenerate the `Form` showing the data the user submitted plus validation errors.

For this example, let's simply make sure they entered something in all the fields. To do that we will create a simple validation function:

```
required :: Strict.Text -> Either AppError Strict.Text
required txt
    | Strict.null txt = Left Required
    | otherwise       = Right txt
```

In this case we are simply checking that the `String` is not null. If it is null we return an error, otherwise we return the `String` unmodified. Some validators will actually transform the value – such as converting the `String` to an `Integer`.

To apply this validation function we can use `transformEither`:

```
transformEither :: Monad m =>
                   Form m input error view anyProof a
                -> (a -> Either error b)
                -> Form m input error view () b
```

We can update our `Form` to:

```
validPostForm :: SimpleForm Message
validPostForm =
    Message <$> name <*> title <*> msg <*  inputSubmit "post"
 where
   name  = errorList ++> labelText "name:"             ++>
           (inputText ""     `transformEither` required)  <++ br

   title = errorList ++> labelText "title:"            ++>
           (inputText ""      `transformEither` required) <++ br
```

```
   msg   = errorList ++> (labelText "message:" <++ br) ++>
             (textarea 80 40 "" `transformEither` required) <++ br
```

The `errorList` will add a list of error messages to a `Form` element. This gives greater control over where error messages appear in the form. The list of errors is literally a list of errors inside a `<ul>` tag:

```
<ul class="reform-error-list">
  <li>error 1</li>
  <li>error 2</li>
  <li>error n</li>
</ul>
```

You can use CSS to control the theming.

For even greater control we could use the `Text.Reform.Generalized.errors` function:

```
errors :: Monad m =>
          ([error] -> view) -- ^ convert the error messages into a view
       -> Form m input error view () ()
```

This allows you to provide your own custom view code for rendering the errors.

We can wrap up the `validForm` the exact same way we did `postForm`:

```
validPage :: AppT IO Response
validPage =
 dir "valid" $
  let action = "/valid" :: Text in
  appTemplate "valid post" () $ [hsx|
   <% reform (form action) "valid" displayMsg Nothing validPostForm %>
  |]
 where
  displayMsg msg =
    appTemplate "Your Message" () $ renderMessage msg
```

A few names have been changed, but everything else is exactly the same.

Separating Validation and Views

One of the primary motivations behind the changes in `digestive-functors 0.3` is allowing developers to separate the validation code from the code which generates the view. We can do this using `reform` as well – in a manner that is both more flexible and which provides greater type safety. The key is the `proof` parameter – which we have so far set to `()` and otherwise ignored.

In `reform` we divide the work into two pieces:

1. `Proofs`

2. a `Form` that returns a `Proved` value

This allows the library authors to create `Proofs` and demand that a `Form` created by another developer satisfies the `Proof`. At the same time, it gives the developer unrestricted control over the layout of the `Form` – including choice of templating library.

Let's create a new type alias for `Form` that allows us to actually set the `proof` parameter:

```
type ProofForm proof =
  Form IO [Input] AppError [AppT IO XML] proof
```

First we will explore the `Proof` related code that would go into a library.

The `proof` parameter for a `Form` is used to indicate that something has been proven about the form's return value.

Two create a `proof` we need two things:

1. a type which names the proof
2. a function which performs the proof

We wrap those two pieces up into a `Proof`:

```
data Proof m error proof a b = Proof
    { proofName     :: proof                       -- ^ name of the
                                                   --   thing to prove
    , proofFunction :: a -> m (Either error b) -- ^ function which
                                                   --   provides the proof
    }
```

In `validPostForm`, we checked that the input fields were not empty `Strings`. We could turn that check into a proof by first creating a type to name that proof:

```
data NotNull = NotNull
```

and then creating a proof function like this:

```
assertNotNull :: (Monad m) =>
                 error
              -> Strict.Text
              -> m (Either error Strict.Text)
assertNotNull errorMsg txt
    | Strict.null txt = return (Left errorMsg)
    | otherwise       = return (Right txt)
```

We can then wrap the two pieces up into a proof:

```
notNullProof :: (Monad m) =>
                error -- ^ error to return if list is empty
             -> Proof m error NotNull Strict.Text Strict.Text
notNullProof errorMsg =
```

```
    Proof { proofName     = NotNull
          , proofFunction = assertNotNull errorMsg
          }
```

We can also create proofs that combine existing proofs. For example, a `Message` is only valid if all its fields are not null. So, first thing we want to do is create a proof name for valid messages:

```
data ValidMessage = ValidMessage
```

The `Message` constructor has the type:

```
Message :: String -> String -> String -> Message
```

For `SimpleForm` we would use `pure` to turn `Message` into a `SimpleForm`:

```
mkSimpleMessage :: SimpleForm (String -> String -> String -> Message)
mkSimpleMessage = pure Message
```

For `ProofForm`, we can do the same thing use `ipure`:

```
mkMessage :: ProofForm (NotNull -> NotNull -> NotNull -> ValidMessage)
                       (Strict.Text -> Strict.Text -> Strict.Text -> Message)
mkMessage = ipure (\NotNull NotNull NotNull -> ValidMessage) Message
```

This creates a chain of validation since `mkMessage` can only be applied to `String` values that have been proven `NotNull`.

The library author can now specify that the user supplied `Form` has the type:

```
someFunc :: ProofForm ValidMessage Message -> ...
```

You will notice that what we have constructed so far has imposes no restrictions on what types of form elements can be used, what template library must be used, or what web server must be used. At the same time, in order for the library user to create a `ProofForm` with the required type, they must apply the supplied validators. Now, clearly a devious library user could use evil tricks to circumvent the system – and they will get what they deserve.

To construct the `Form`, we use a pattern very similar to what we did when using `SimpleForm`. They only real differences are:

1. we use `prove` instead of `transformEither`
2. we use `<<*>>` instead of `<*>`

To apply a `Proof` we use the `prove` function:

```
prove :: (Monad m) =>
         Form m input error view q a
      -> Proof m error proof a b
      -> Form m input error view proof b
```

So, we can make a `ProofForms` for non-empty `Strings` like this:

```
inputText' :: Strict.Text -> ProofForm NotNull Strict.Text
inputText' initialValue =
    inputText initialValue `prove` (notNullProof Required)

textarea' :: Int         -- ^ cols
          -> Int         -- ^ rows
          -> Strict.Text -- ^ initial value
          -> ProofForm NotNull Strict.Text
textarea' cols rows initialValue =
    textarea cols rows initialValue `prove` (notNullProof Required)
```

to create the `ValidMessage` form we can then combine the pieces like:

```
provenPostForm :: ProofForm ValidMessage Message
provenPostForm =
    mkMessage
      <<*>> errorList ++> labelText "name: "    ++> inputText' ""
      <<*>> errorList ++> labelText "title: "   ++> inputText' ""
      <<*>> errorList ++> labelText "message: " ++> textarea' 80 40 ""
```

This code looks quite similar to our `validPostForm` code. The primary difference is that we use `<<*>>` instead of `<*>`. That brings is to the topic of type-indexed applicative functors.

Type Indexed / Parameterized Applicative Functors

Lets look at the type for `Form` again:

```
newtype Form m input error view proof a = Form { ... }
```

In order to make an `Applicative` instance of `Form`, all the proof type variables must be the same type and must form a `Monoid`:

```
instance (Functor m, Monad m, Monoid view, Monoid proof) =>
             (Form m input error view proof)
```

for `SimpleForm` we used the following instance, which is defined for us already in `reform`:

```
instance (Functor m, Monoid view, Monad m) =>
             Applicative (Form m input error view ())
```

With this instance, `reform` feels and works almost exactly like `digestive-functors <= 0.2`.

But, for the `provenPostForm`, that `Applicative` instance won't work for us. `mkMessage` has the type:

```
mkMessage :: ProofForm (NotNull -> NotNull -> NotNull -> ValidMessage)
                       (String -> String -> String -> Message)
```

and we want to apply it to `ProofForms` created by:

```
inputText' :: String -> ProofForm NotNull String
```

Here the proof types don't match up. Instead we need a `Applicative Functor` that allows us to transform the return value *and* the proof value. We need, what I believe is called, a `Type-Indexed Applicative Functor` or a `Parameterized Applicative Functor`. Most literature on this subject is actually dealing with type-indexed or parameterized `Monads`, but the idea is the same.

The `reform` library defines two new classes, `IndexedFunctor` and `IndexedApplicative`:

```
class IndexedFunctor f where
    -- | imap is similar to fmap
    imap :: (x -> y) -- ^ function to apply to first parameter
         -> (a -> b) -- ^ function to apply to second parameter
         -> f x a    -- ^ indexed functor
         -> f y b

class (IndexedFunctor f) => IndexedApplicative f where
    -- | similar to 'pure'
    ipure   :: x -> a -> f x a
    -- | similar to '<*>'
   (<<*>>) :: f (x -> y) (a -> b) -> f x a -> f y b
```

These classes look just like their non-indexed counterparts, except that they transform an extra parameter. Now we can create instances like:

```
instance (Monad m)              =>
   IndexedFunctor     (Form m input view error) where

instance (Monad m, Monoid view) =>
   IndexedApplicative (Form m input error view) where
```

We use these classes the same way we would use the normal `Functor` and `Applicative` classes. The only difference is that the type-checker can now enforce the proofs.

Using `Proofs` in unproven `Forms`

The `Proof` module provides a handful of useful `Proofs` that perform transformations, such as converting a `String` to a `Int`:

```
decimal :: (Monad m, Eq i, Num i) =>
           (Text -> error) -- ^ create an error message
```

```
                              --   ('Text' is the value
                              --   that did not parse)
        -> Proof m error Decimal String i
```

We can use this `Proof` with our `SimpleForm` by using the `transform` function:

```
transform :: (Monad m) =>
             Form m input error view anyProof a
          -> Proof m error proof a b
          -> Form m input error view () b
```

`transform` is similar to the `prove` function, except it ignores the proof name and sets the proof to `()`. Technically `()` is still a proof – but we consider it to be the proof that proves nothing.

Here is an example of using `transform` with `decimal` to create a simple form that parses a positive `Integer` value:

```
inputInteger :: SimpleForm Integer
inputInteger = inputText "" `transform` (decimal NotANatural)
```

Conclusion

And, that is the essence of `reform`. The Haddock documentation should cover the remainder – such as other types of input controls (radio buttons, checkboxes, etc).

main

Here is a main function that ties all the examples together:

```
main :: IO ()
main = simpleHTTP nullConf $ unHSPT $ unXMLGenT $ do
 decodeBody (defaultBodyPolicy "/tmp/" 0 10000 10000)
 msum [ postPage
      , postPage2
      , validPage
      , do nullDir
           appTemplate "forms" () $ [hsx|
            <ul>
             <li><a href="/post">Simple Form</a></li>
             <li>
               <a href="/post2">
                 Simple Form (postPage2 implementation)
               </a>
             </li>
             <li><a href="/valid">Valid Form</a></li>
```

```
            </ul> |]
      ]
```

There is nothing `reform` specific about it.

Source code for the app is here.

web-routes

The `web-routes` libraries provide a system for type-safe url routing. The basic concept behind type-safe urls is very simple. Instead of working directly with url as strings, we create a type that represents all the possible urls in our web application. By using types instead of strings we benefit in several ways:

fewer runtime errors due to typos If you mistype `"/hmoe"` instead of `"/home"`, the compiler will gleefully compile it. But if you mistype the constructor as `Hmoe` instead of `Home` you will get a compile time error.

Compile type assurance that all routes are mapped to handlers Routing is performed via a simple `case` statement on the url type. If you forget to handle a route, the compiler will give you a `Pattern match(es) are non-exhaustive` warning.

unique URLs for 3rd party libraries Libraries (such as a blog or image gallery component) need a safe way to create urls that do no overlap with the routes provided by other libraries. For example, if a blog component and image component both try to claim the url `/upload`, something bad is going to happen. With `web-routes`, libraries do not have to take any special steps to ensure that the urls they generate are unique. `web-routes` are composable and result in unique urls.

Compile time errors when routes change As a website evolves, existing routes might change or be removed entirely. With `web-routes` this will result in a change to the type. As a result, code that has not been updated will generate a compile-time error, instead of a runtime error. This is especially valuable when using 3rd party libraries, since you may not even be aware that the route had changed otherwise.

better separation of presentation and behavior In `web-routes`, the parsing and printing of a url is separated from the mapping of a url to a handler or creating hyperlinks in your code. This makes it trivial to change the way the url type is converted to a string and back. You need only modify the function that does the conversion, and everything else can stay the same. You do not need to hunt all over the code trying to find places that use the old format.

automatic sitemap Because the url type represents all the valid routes on your site, it also acts as a simple sitemap.

`web-routes` is designed to be very flexible. For example, it does not require that you use any particular mechanism for defining the mapping between the url type and the url string. Instead, we provide a variety of addon packages that provide different methods including, template-haskell, generics, parsec, quasi-quotation, and more. This means it is also easy to add your own custom mechanism. For example, you might still use template-haskell, but with a different set of rules for converting a type to a string.

`web-routes` is also not limited to use with any particular framework, templating system, database, etc.

`web-routes` Demo

Let's start by looking at a simple example of using `web-routes`. In this example we will use blaze for the HTML templates.

In order to run this demo you will need to install `web-routes`, `web-routes-th` and `web-routes-happstack` from hackage.

```
{-# LANGUAGE DeriveDataTypeable, GeneralizedNewtypeDeriving,
    TemplateHaskell #-}
module Main where

import Prelude                 hiding (head)

import Control.Monad           (msum)
import Data.Data               (Data, Typeable)
import Data.Monoid             (mconcat)
import Data.Text               (pack)
import Happstack.Server
    ( Response, ServerPartT, ok, toResponse, simpleHTTP
    , nullConf, seeOther, dir, notFound, seeOther)
import Text.Blaze.Html4.Strict
    ( Html, (!), html, head, body, title, p, toHtml
    , toValue, ol, li, a)
import Text.Blaze.Html4.Strict.Attributes (href)
import Web.Routes
    ( PathInfo(..), RouteT, showURL
    , runRouteT, Site(..), setDefault, mkSitePI)
import Web.Routes.TH           (derivePathInfo)
import Web.Routes.Happstack    (implSite)
```

First we need to define the type to represent our routes. In this site we will have a homepage and articles which can be retrieved by their id.

```
newtype ArticleId = ArticleId { unArticleId :: Int }
    deriving (Eq, Ord, Enum, Read, Show, Data, Typeable, PathInfo)

data Sitemap
    = Home
    | Article ArticleId
    deriving (Eq, Ord, Read, Show, Data, Typeable)
```

Next we use `template-haskell` to derive an instance of `PathInfo` for the `Sitemap` type.

```
$(derivePathInfo ''Sitemap)
```

The `PathInfo` class is defined in `Web.Routes` and looks like this:

```
class PathInfo a where
  toPathSegments :: a -> [Text]
  fromPathSegments :: URLParser a
```

It is basically a class that describes how to turn a type into a URL and back. This class is semi-optional. Some conversion methods such as `web-routes-th` and `web-routes-regular` use it, but others do not.

Since `ArticleId` is just a `newtype` we were able to just do `deriving PathInfo` instead of having to call `derivePathInfo`.

Next we need a function that maps a route to the handlers:

```
route :: Sitemap -> RouteT Sitemap (ServerPartT IO) Response
route url =
    case url of
      Home                -> homePage
      (Article articleId) -> articlePage articleId
```

As you can see, mapping a URL to a handler is just a straight-forward case statement. We do not need to do anything fancy here to extract the article id from the URL, becuse that has already been done when the URL was converted to a `Sitemap` value.

You may be tempted to write the `route` function like this instead of using the case statement:

```
route :: Sitemap -> RouteT Sitemap (ServerPartT IO) Response
route Home                = homePage
route (Article articleId) = articlePage articleId
```

But, I don't recommend it. In a real application, the `route` function will likely take a number of extra arguments such as database handles. Every time you add a parameter, you have to update every pattern match to account for the extra argument, even for the handlers that don't use it. Using a `case` statement instead makes the code easier to maintain and more readable in my opinion.

The other thing you will notice is the `RouteT` monad transformer in the type signature. The `RouteT` monad transformer is another semi-optional feature of `web-routes`. `RouteT` is basically a `Reader` monad that holds the function which converts the URL type into a string. At first, this seems unnecessary – why not just call `toPathInfo` directly and skip `RouteT` entirely? But it turns out there are few advantages that `RouteT` brings:

1. `RouteT` is parametrized by the URL type – in this case `Sitemap`. That will prevent us from accidentally trying to convert an `ArticleId` into a URL. An `ArticleId` is a valid component of some URLs, but it is not a valid URL by itself.
2. The URL showing function inside `RouteT` can also contain additional information needed to form a valid URL, such as the hostname name, port, and path prefix
3. `RouteT` is also used when we want to embed a library/sub-site into a larger site.

We will see examples of these benefits as we continue with the tutorial.

Next, we have the handler functions:

```
homePage :: RouteT Sitemap (ServerPartT IO) Response
homePage = do
  articles <- mapM mkArticle [(ArticleId 1) .. (ArticleId 10)]
  ok $ toResponse $
    html $ do
      head $ title $ (toHtml "Welcome Home!")
      body $ do
        ol $ mconcat articles
  where
    mkArticle :: ArticleId -> RouteT Sitemap (ServerPartT IO) Html
    mkArticle articleId =
        do url <- showURL (Article articleId)
           return $ li $ a ! href (toValue url) $
              toHtml $ "Article " ++ (show $ unArticleId articleId)

articlePage :: ArticleId -> RouteT Sitemap (ServerPartT IO) Response
articlePage (ArticleId articleId) = do
  homeURL <- showURL Home
  ok $ toResponse $
     html $ do
       head $ title $ (toHtml $ "Article " ++ show articleId)
       body $ do
        p $ do toHtml $ "You are now reading article "
               toHtml $ show articleId
        p $ do toHtml "Click "
               a ! href (toValue homeURL) $ toHtml "here"
```

```
                toHtml " to return home."
```

Even though we have the `RouteT` in the type signature – these functions look normal `ServerPartT` functions – we do not have to use `lift` or anything else. That is because `RouteT` is a instance of all the `Happstack` class and the related classes such as `ServerMonad, FilterMonad,` etc. Though you do need to make sure you have imported `Web.Routes.Happstack` to get those instances.

The only new thing here is the `showURL` function, which has the type:

```
showURL :: ShowURL m => URL m -> m Text
```

`showURL` converts a url type, such as `Sitemap` into `Text` that we can use as an attribute value for an `href`, `src`, etc.

`URL m` is a type-function that calculates the type based on the monad we are currently in. For `RouteT url m a`, `URL m` is going to be whatever `url` is. In this example, `url` is `Sitemap`. If you are not familiar with type families and type functions, see section of `web-routes and type-families`.

Now we have:

1. A url type, `Sitemap`
2. functions to convert the type to a string and back via `PathInfo`
3. a function to route the url to a handler, `route`

We need to tie these three pieces together. That is what the `Site` type does for us:

```
data Site url a = Site {
   -- | function which routes the url to a handler
   handleSite         :: (url -> [(Text, Text)] -> Text) -> url -> a
   -- | This function must be the inverse of 'parsePathSegments'.
 , formatPathSegments :: url -> ([Text], [(Text, Text)])
   -- | This function must be the inverse of 'formatPathSegments'.
 , parsePathSegments  :: [Text] -> Either String url
}
```

Looking at the type for `Site`, we notice that it is very general – it does not have any references to `Happstack`, `PathInfo`, `URLParser`, `RouteT`, etc. That is because those are all addons to the core of `web-routes`. We can convert our `route` to a `Site` using some simple helper functions like this:

```
site :: Site Sitemap (ServerPartT IO Response)
site =
       setDefault Home $ mkSitePI (runRouteT route)
```

`runRouteT` removes the `RouteT` wrapper from our routing function:

```
runRouteT :: (url -> RouteT url m a)
          -> ((url -> [(Text, Text)] -> Text) -> url -> m a)
```

So if we have our routing function like:

```
route :: Sitemap
      -> RouteT Sitemap (ServerPartT IO) Response
```

`runRouteT` will convert it to a function that takes a url showing function:

```
(runRouteT route) :: (Sitemap -> [(Text, Text)] -> Text)
                  -> Sitemap
                  -> ServerPartT IO Response
```

Since we created a `PathInfo` instance for `Sitemap` we can use `mkSitePI` to convert the new function to a `Site`. `mkSitePI` has the type:

```
mkSitePI :: (PathInfo url) =>
            ((url -> [(Text, Text)] -> Text) -> url -> a)
         -> Site url a
```

so applying it to `runRouteT route` gives us:

```
(mkSitePI (runRouteT route)) :: Site Sitemap (ServerPartT IO Response)
```

`setDefault` allows you to map `/` to any route you want. In this example we map `/` to `Home`.

```
setDefault :: url -> Site url a -> Site url a
```

Next we use `implSite` to embed the `Site` into a normal Happstack route:

```
main :: IO ()
main = simpleHTTP nullConf $ msum
  [ dir "favicon.ico" $ notFound (toResponse ())
  , implSite (pack "http://localhost:8000") (pack "/route") site
  , seeOther "/route" (toResponse ())
  ]
```

The type for `implSite` is straight-forward:

```
implSite :: (Functor m, Monad m, MonadPlus m, ServerMonad m) =>
            Text           -- ^ "http://example.org"
         -> FilePath       -- ^ path to this handler, .e.g. "/route"
         -> Site url (m a) -- ^ the 'Site'
         -> m a
```

The first argument is the domain/port/etc that you want to add to the beginning of any URLs you show. The first argument is not used during the decoding/routing process – it is merely prepended to any generated url strings.

The second argument is the path to this handler. This path is automatically used when routing the incoming request and when showing the URL. This path can be used to ensure that all routes generated by `web-routes` are unique because they will be in a separate sub-directory (aka, a separate namespace). If you do

not want to put the routes in a separate sub-directory you can set this field to "".

The third argument is the `Site` that does the routing.

If the URL decoding fails, then `implSite` will call `mzero`.

Sometimes you will want to know the exact parse error that caused the router to fail. You can get the error by using `implSite_` instead. Here is an alternative `main` that prints the route error to `stdout`.

```
main :: IO ()
main = simpleHTTP nullConf $ msum
 [ dir "favicon.ico" $ notFound (toResponse ())
 , do r <- implSite_ (pack "http://localhost:8000") (pack "/route") site
      case r of
        (Left e) -> liftIO (print e) >> mzero
        (Right m) -> return m
 , seeOther "/route" (toResponse ())
 ]
```

Source code for the app is here.

web-routes + Type Families

`showURL` has the type:

```
showURL :: ShowURL m => URL m -> m Text
```

If you are not familiar with type families and type functions, the `URL m` in that type signature might look a bit funny. But it is really very simple.

The `showURL` function leverages the `ShowURL` class:

```
class ShowURL m where
   type URL m
   showURLParams :: (URL m) -> [(Text, Text)] -> m Text
```

And here is the `RouteT` instance for `ShowURL`:

```
instance (Monad m) => ShowURL (RouteT url m) where
   type URL (RouteT url m) = url
   showURLParams url params =
       do showF <- askRouteT
          return (showF url params)
```

Here `url` is a *type function* that is applied to a type and gives us another type. For example, writing `URL (RouteT Sitemap (ServerPartT IO))` gives us the type `Sitemap`. We can use the type function any place we would normally use a type.

In our example we had:

```
homeURL <- showURL Home
```

So there, showURL is going to have the type:

```
showURL :: URL (RouteT Sitemap (ServerPartT IO))
        -> RouteT Sitemap (ServerPartT IO) Text
```

which can be simplified to:

```
showURL :: Sitemap -> RouteT Sitemap (ServerPartT IO) Text
```

So, we see that the URL type we pass to `showURL` is dictated by the monad we are currently in. This ensures that we only call `showURL` on values of the right type.

While `ShowURL` is generally used with the `RouteT` type – it is not actually a requirement. You can implement `ShowURL` for any monad of your choosing.

web-routes-boomerang

In the previous example we used Template Haskell to automatically derive a mapping between the URL type and the URL string. This is very convenient early in the development process when the routes are changing a lot. But sometimes we want more precise control over the look of the URL. One solution is to write the mappings from the URL type to the URL string by hand.

One way to do that would be to write one function to show the URLs, and another function that uses `parsec` to parse the URLs. But having to say the same thing twice is really annoying and error prone. What we really want is a way to write the mapping once, and automatically exact a parser and printer from the specification.

Fortunately, Sjoerd Visscher and Martijn van Steenbergen figured out exactly how to do that and published a proof of concept library know as `Zwaluw`. With permission, I have refactored their original library into two separate libraries: `boomerang` and `web-routes-boomerang`.

The technique behind `Zwaluw` and `Boomerang` is very cool. But in this tutorial we will skip the theory and get right to the practice.

In order to run this demo you will need to install `web-routes`, `web-routes-boomerang` and `web-routes-happstack` from hackage.

We will modify the previous demo to use `boomerang` in order to demonstrate how easy it is to change methods midstream. We will also add a few new routes to demonstrate some features of using `boomerang`.

```
{-# LANGUAGE DeriveDataTypeable, GeneralizedNewtypeDeriving
  , TemplateHaskell, TypeOperators, OverloadedStrings #-}
```

```
module Main where
```

The first thing to notice is that we hide `id` and `(.)` from the `Prelude` and import the versions from `Control.Category` instead.

```
import Prelude                 hiding (head, id, (.))
import Control.Category        (Category(id, (.)))

import Control.Monad           (msum)
import Data.Data               (Data, Typeable)
import Data.Monoid             (mconcat)
import Data.String             (fromString)
import Data.Text               (Text)
import Happstack.Server
    ( Response, ServerPartT, ok, toResponse, simpleHTTP
    , nullConf, seeOther, dir, notFound, seeOther)
import Text.Blaze.Html4.Strict
    ( Html, (!), html, head, body, title, p, toHtml
    , toValue, ol, li, a)
import Text.Blaze.Html4.Strict.Attributes (href)
import Text.Boomerang.TH       (makeBoomerangs)
import Web.Routes
    ( PathInfo(..), RouteT, showURL
    , runRouteT, Site(..), setDefault, mkSitePI)
import Web.Routes.TH           (derivePathInfo)
import Web.Routes.Happstack    (implSite)
import Web.Routes.Boomerang
```

Next we have our `Sitemap` types again. `Sitemap` is similar to the previous example, except it also includes `UserOverview` and `UserDetail`.

```
newtype ArticleId = ArticleId { unArticleId :: Int }
    deriving (Eq, Ord, Enum, Read, Show, Data, Typeable, PathInfo)

data Sitemap
    = Home
    | Article ArticleId
    | UserOverview
    | UserDetail Int Text
    deriving (Eq, Ord, Read, Show, Data, Typeable)
```

Next we call `makeBoomerangs`:

```
$(makeBoomerangs ''Sitemap)
```

That will create new combinators corresponding to the constructors for `Sitemap`. They will be named, `rHome`, `rArticle`, `rUserOverview`, and `rUserDetail`. These combinators are used to apply/remove the corresponding constructors, but they do not affect the appearance of the route at all. You could create these

helper functions by hand, but they are dreadful boring and there is no advantage to doing so.

Now we can specify how the `Sitemap` type is mapped to a URL string and back:

```
sitemap :: Router () (Sitemap :- ())
sitemap =
    (  rHome
    <> rArticle . (lit "article" </> articleId)
    <> lit "users" . users
    )
    where
      users =  rUserOverview
            <> rUserDetail </> int . lit "-" . anyText

articleId :: Router () (ArticleId :- ())
articleId =
    xmaph ArticleId (Just . unArticleId) int
```

The mapping looks like this:

URL	<=>	type
/	<=>	`Home`
/article/*int*	<=>	`Article Int`
/users/	<=>	`UserOverview`
/users/*int-string*	<=>	`UserDetail Int String`

The `sitemap` function looks like an ordinary parser. But, what makes it is exciting is that it also defines the pretty-printer at the same time.

By examining the mapping table and comparing it to the code, you should be able to get an intuitive feel for how `boomerang` works. The key boomerang features we see are:

`<>` is the choice operator. It chooses between the various paths. `.`
: is used to combine elements together. `</>`
: matches on the / between path segments. (The combinators, such as `lit`, `int`, `anyText`, operate on a single path segment.) `lit`
: matches on a string literal. If you enabled `OverloadedStrings` then you do not need to explicitly use the `lit` function. For example, you could just write, `int . "-" . anyText`. `int`
: matches on an `Int`. `anyText`
: matches on any string. It keeps going until it reaches the end of the current path segment. `xmaph`
: is a bit like `fmap`, except instead of only needing `a -> b` it also needs the other direction, `b -> Maybe a`.

```
xmaph :: (a -> b)
      -> (b -> Maybe a)
      -> Boomerang e tok i (a :- o)
      -> Boomerang e tok i (b :- o)
```

In this example, we use `xmaph` to convert `int :: Router () (Int :- ())` into `articleId :: Router () (ArticleId :- ())`.

longest route You will notice that the parser for /users comes before /users/*int-string*. Unlike `parsec`, the order of the parsers (usually) does not matter. We also do not have to use `try` to allow for backtracking. `boomerang` will find all valid parses and pick the best one. Here, that means the parser that consumed the most available input.

`Router` type is just a simple alias:

```
type Router a b = Boomerang TextsError [Text] a b
```

Looking at this line:

```
            <> rUserDetail </> int . lit "-" . anyText
```

and comparing it to the constructor

```
    UserDetail Int Text
```

we see that the constructor takes two arguments, but the mapping uses three combinators, `int`, `lit`, and `anyText`. It turns out that some combinators produce/consume values from the URL type, and some do not. We can find out which do and which don't by looking at the their types:

```
int     ::          Boomerang TextsError [Text] r (Int :- r)
anyText ::          Boomerang TextsError [Text] r (Text :- r)
lit     :: Text ->  Boomerang TextsError [Text] r r
```

We see `int` takes `r` and produces `(Int :- r)` and `anyText` takes `r` and produces `(Text :- r)`. While `lit` takes `r` and returns `r`.

Looking at the type of the all three composed together we get:

```
int . lit "-" . anyText :: Boomerang TextsError [Text] a (Int :- (Text :- a))
```

So there we see the `Int` and `Text` that are arguments to `UserDetail`.

Looking at the type of `rUserDetail`, we will see that it has the type:

```
 rUserDetail :: Boomerang e tok (Int :- (Text :- r)) (Sitemap :- r)
```

So, it takes an `Int` and `Text` and produces a `Sitemap`. That mirrors what the `UserDetail` constructor itself does:

```
ghci> :t UserDetail
UserDetail :: Int -> Text -> Sitemap
```

Next we need a function that maps a route to the handlers. This is the same exact function we used in the previous example extended with the additional routes:

```
route :: Sitemap -> RouteT Sitemap (ServerPartT IO) Response
route url =
    case url of
      Home                  -> homePage
      (Article articleId)   -> articlePage articleId
      UserOverview          -> userOverviewPage
      (UserDetail uid name) -> userDetailPage uid name
```

Next, we have the handler functions. These are also exactly the same as the previous example, plus the new routes:

```
homePage :: RouteT Sitemap (ServerPartT IO) Response
homePage = do
  articles     <- mapM mkArticle [(ArticleId 1) .. (ArticleId 10)]
  userOverview <- showURL UserOverview
  ok $ toResponse $
    html $ do
     head $ title $ "Welcome Home!"
     body $ do
      a ! href (toValue userOverview) $ "User Overview"
      ol $ mconcat articles
  where
   mkArticle :: ArticleId -> RouteT Sitemap (ServerPartT IO) Html
   mkArticle articleId = do
    url <- showURL (Article articleId)
    return $ li $ a ! href (toValue url) $
      toHtml $ "Article " ++ (show $ unArticleId articleId)

articlePage :: ArticleId
            -> RouteT Sitemap (ServerPartT IO) Response
articlePage (ArticleId articleId) = do
  homeURL <- showURL Home
  ok $ toResponse $
    html $ do
     head $ title $ (toHtml $ "Article " ++ show articleId)
     body $ do
      p $ toHtml $ "You are now reading article " ++ show articleId
      p $ do "Click "
             a ! href (toValue homeURL) $ "here"
             " to return home."

userOverviewPage :: RouteT Sitemap (ServerPartT IO) Response
userOverviewPage = do
  users <- mapM mkUser [1 .. 10]
```

```
  ok $ toResponse $
    html $ do
      head $ title $ "Our Users"
      body $ do
        ol $ mconcat users
  where
    mkUser :: Int -> RouteT Sitemap (ServerPartT IO) Html
    mkUser userId = do
      url <- showURL (UserDetail userId
                      (fromString $ "user " ++ show userId))
      return $ li $ a ! href (toValue url) $
        toHtml $ "User " ++ (show $ userId)

userDetailPage :: Int
               -> Text
               -> RouteT Sitemap (ServerPartT IO) Response
userDetailPage userId userName = do
  homeURL <- showURL Home
  ok $ toResponse $
    html $ do
      head $ title $ (toHtml $ "User " <> userName)
      body $ do
        p $ toHtml $ "You are now view user detail page for " <> userName
        p $ do "Click "
               a ! href (toValue homeURL) $ "here"
               " to return home."
```

Creating the `Site` type is similar to the previous example. We still use `runRouteT` to unwrap the `RouteT` layer. But now we use `boomerangSite` to convert the `route` function into a `Site`:

```
site :: Site Sitemap (ServerPartT IO Response)
site =
  setDefault Home $ boomerangSite (runRouteT route) sitemap
```

The route function is essentially the same in this example and the previous example – it did not have to be changed to work with `boomerang` instead of `PathInfo`. It is the `formatPathSegments` and `parsePathSegments` functions bundled up in the `Site` that change. In the previous example, we used `mkSitePI`, which leveraged the `PathInfo` instances. Here we use `boomerangSite` which uses the `sitemap` mapping we defined above.

The practical result is that you can start by using `derivePathInfo` and avoid having to think about how the URLs will look. Later, once the routes have settled down, you can then easily switch to using `boomerang` to create your route mapping.

Next we use `implSite` to embed the `Site` into a normal Happstack route:

```
main :: IO ()
main = simpleHTTP nullConf $ msum
  [ dir "favicon.ico" $ notFound (toResponse ())
  , implSite "http://localhost:8000" "/route" site
  , seeOther ("/route/" :: String) (toResponse ())
  ]
```

Source code for the app is here.

In this example, we only used a few simple combinators. But `boomerang` provides a whole range of combinators such as `many`, `some`, `chain`, etc. For more information check out the haddock documentation for `boomerang`. Especially the `Text.Boomerang.Combinators` and `Text.Boomerang.Texts` modules.

web-routes and HSP

You will need to install the optional `web-routes`, `web-routes-th`, `web-routes-hsp` and `happstack-hsp` packages for this section.

```
{-# LANGUAGE TemplateHaskell, QuasiQuotes, OverloadedStrings #-}
module Main where

import Control.Applicative ((<$>))
import Control.Monad       (msum)
import Happstack.Server
import Happstack.Server.HSP.HTML
import HSP
import Language.Haskell.HSX.QQ
import Web.Routes
import Web.Routes.TH
import Web.Routes.XMLGenT
import Web.Routes.Happstack
```

If you are using `web-routes` and `HSP` then inserting URLs is especially clean and easy. If we have the URL:

```
data SiteURL = Monkeys Int deriving (Eq, Ord, Read, Show)

$(derivePathInfo ''SiteURL)
```

then we can insert it into some HTML like this:

```
monkeys :: Int -> RouteT SiteURL (ServerPartT IO) Response
monkeys n =
    do html <- defaultTemplate "monkeys" () $ [hsx|
        <%>
         You have <% show n %> monkeys.
         Click <a href=(Monkeys (succ n))>here</a> for more.
```

```
        </%> |]
       ok $ (toResponse html)
```

Notice in particular this bit:

```
<a href=(Monkeys (succ n))>here</a>
```

We do not need `showURL`, we just use the URL type directly. That works because `Web.Routes.XMLGenT` provides an instance:

```
instance (Functor m, Monad m) =>
    EmbedAsAttr (RouteT url m) (Attr Text url)
```

Here is the rest of the example:

```
route :: SiteURL -> RouteT SiteURL (ServerPartT IO) Response
route url =
    case url of
      (Monkeys n) -> monkeys n

site :: Site SiteURL (ServerPartT IO Response)
site = setDefault (Monkeys 0) $ mkSitePI (runRouteT route)

main :: IO ()
main = simpleHTTP nullConf $
  msum [ dir "favicon.ico" $ notFound (toResponse ())
       , implSite "http://localhost:8000" "" site
       ]
```

Source code for the app is here.

acid-state

`acid-state` is a NoSQL, RAM-cloud, persistent data store. One attractive feature is that it's designed to store arbitrary Haskell datatypes and queries are written using plain old Haskell code. This means you do not have to learn a special query language, or figure out how to turn your beautiful Haskell datastructures into some limited set of ints and strings.

`acid-state` and `safecopy` are the successors to the old `happstack-state` and `happstack-data` libraries. You can learn more at the acid-state homepage. `acid-state` is now completely independent from Happstack and can be used with any web framework. However, Happstack is still committed to the improvement and promotion of `acid-state`.

Apps written using `happstack-state` can be migrated to use `acid-state` relatively easily. Details on the process or documented here.

How `acid-state` works

A very simple way to model a database in Haskell would be to create a datatype to represent your data and then store that data in a mutable, global variable, such as a global `IORef`. Then you could just write normal Haskell functions to query that value and update it. No need to learn a special query language. No need to marshal your types from expressive Haskell datatypes to some limited set of types supported by an external database.

That works great.. as long as your application is only single-threaded, and as long as it never crashes, and never needs to be restarted. For a web application, those requires are completely unacceptable – but the idea is still appealing. `acid-state` provides a practical implementation of that idea which actually implements the ACID guarantees that you may be familiar with from traditional relational databases such as MySQL, postgres, etc.

In `acid-state` we start by defining a type that represents the state we wish to store. Then we write a bunch of pure functions that query that value or which return an updated value. However, we do not call those functions directly. Instead we keep the value inside an `AcidState` handle, and we call our functions

indirectly by using the `update` and `query` functions. This allows `acid-state` to transparently log update events to disk, to ensure that update and query events run automatically and in isolation, etc. It is allows us to make remote API calls, and, eventually, replication and multimaster.

Note that when we say `acid-state` is pure, we are referring specifically to the fact that the functions we write to perform updates and queries are pure. `acid-state` itself must do IO in order to coordinate events from multiple threads, log events to disk, perform remote queries, etc.

Now that you have a vague idea how `acid-state` works, let's clarify it by looking at some examples.

acid-state counter

Our first example is a very simple hit counter app.

First a bunch of `LANGUAGE` pragmas and imports:

```
{-# LANGUAGE CPP, DeriveDataTypeable, FlexibleContexts,
  GeneralizedNewtypeDeriving, MultiParamTypeClasses,
  TemplateHaskell, TypeFamilies, RecordWildCards #-}

module Main where

import Control.Applicative  ( (<$>) )
import Control.Exception    ( bracket )
import Control.Monad        ( msum )
import Control.Monad.Reader ( ask )
import Control.Monad.State  ( get, put )
import Data.Data            ( Data, Typeable )
import Happstack.Server     ( Response, ServerPart, dir
                            , nullDir, nullConf, ok
                            , simpleHTTP, toResponse )
import Data.Acid            ( AcidState, Query, Update
                            , makeAcidic, openLocalState )
import Data.Acid.Advanced   ( query', update' )
import Data.Acid.Local      ( createCheckpointAndClose )
import Data.SafeCopy        ( base, deriveSafeCopy )
```

Next we define a type that we wish to store in our state. In this case we just create a simple record with a single field `count`:

```
data CounterState = CounterState { count :: Integer }
    deriving (Eq, Ord, Read, Show, Data, Typeable)

$(deriveSafeCopy 0 'base ''CounterState)
```

`deriveSafeCopy` creates an instance of the `SafeCopy` class for `CounterState`. `SafeCopy` is class for versioned serialization, deserilization, and migration. The `SafeCopy` class is a bit like a combination of the `Read` and `Show` classes, except that it converts the data to a compact `ByteString` representation, and it includes version information in case the type changes and old data needs to be migrated.

Since this is the first version of the `CounterState` type, we give it version number 0 and declare it to be the `base` type. Later if we change the type, we would increment the version to 1 and declare it to be an `extension` of a previous type. We would also provide a migration instance to migrate the old type to the new type. The migration would happen automatically when the old state is read. For more information on `SafeCopy, base, extension` and migration see the haddock docs. (A detailed section on migration for the Crash Course is planned, but not yet written).

If you are not familiar with Template Haskell be sure to read the Template Haskell appendix for brief intro to Template Haskell.

Next we will define an initial value that is suitable for initializing the `CounterState` state.

```
initialCounterState :: CounterState
initialCounterState = CounterState 0
```

Now that we have our types, we can define some update and query functions.

First let's define an update function which increments the count and returns the incremented value:

```
incCountBy :: Integer -> Update CounterState Integer
incCountBy n =
    do c@CounterState{..} <- get
       let newCount = count + n
       put $ c { count = newCount }
       return newCount
```

In this line:

```
c@CounterState{..} <- get
```

we are using the `RecordWildCards` extension. The `{..}` binds all the fields of the record to symbols with the same name. That is why in the next line we can just write `count` instead of `(count c)`. Using `RecordWildCards` here is completely optional, but tends to make the code less cluttered, and easier to read.

Also notice that we are using the `get` and `put` functions from `MonadState` to get and put the ACID state. The `Update` monad is basically an enchanced version of the `State` monad. For the moment it is perhaps easiest to just pretend that `incCountBy` has the type signature:

```
incCountBy :: Integer -> State CounterState Integer
```

And then it becomes clearer that `incCountBy` is just a simple function in the `State` monad which updates `CounterState` and returns an `Integer`.

Note that even though we are using a monad here.. the code is still pure. If we wanted we could have required the update function to have a type like this instead:

```
incCountBy :: Integer -> CounterState -> (CounterState, Integer)
```

In that version, the current state is explicitly passed in, and the function explicitly returns the updated state. The monadic version does the same thing, but uses `>>=` to make the plumbing easier. This makes the monadic version easier to read and reduces mistakes.

When we later use the `update` function to call `incCountBy`, `incCountBy` will be run in an isolated manner (the 'I' in ACID). That means that you do not need to worry about some other thread modifying the `CounterState` between the `get` and the `put`. It will also be run atomically (the 'A' in ACID), meaning that either the whole function will run or it will not run at all. If the server is killed mid-transaction, the transaction will either be completely applied or not applied at all.

You may also note that `Update` (and `State`) are not instances of the `MonadIO` class. This means you can not perform IO inside the update. This is by design. In order to ensure Durability and to support replication, events need to be pure. That allows us to be confident that if the event log has to be replayed – it will result in the same state we had before.

We can also define a query which reads the state, and does not update it:

```
peekCount :: Query CounterState Integer
peekCount = count <$> ask
```

The `Query` monad is an enhanced version of the `Reader` monad. So we can pretend that `peekCount` has the type:

```
peekCount :: Reader CounterState Integer
```

Although we could have just used `get` in the `Update` monad, it is better to use the `Query` monad if you are doing a read-only operation because it will not block other database transactions. It also lets the user calling the function know that the database will not be affected.

Next we have to turn the update and query functions into acid-state events. This is almost always done by using the Template Haskell function `makeAcidic`

```
$(makeAcidic ''CounterState ['incCountBy, 'peekCount])
```

The `makeAcidic` function creates a bunch of boilerplate types and type class instances. If you want to see what is happening under the hood, check out the examples here. The examples with names like, `HelloWorldNoTH.hs` show how to implement the boilerplate by hand. In practice, you will probably never want

to or need to do this. But you may find it useful to have a basic understanding of what is happening. You could also use the `-ddump-splices` flag to GHC to see the auto-generated instances – but the lack of formatting makes it difficult to read.

Here we actually call our query and update functions:

```
handlers :: AcidState CounterState -> ServerPart Response
handlers acid = msum
  [ dir "peek" $ do
      c <- query' acid PeekCount
      ok $ toResponse $"peeked at the count and saw: " ++ show c
  , do nullDir
       c <- update' acid (IncCountBy 1)
       ok $ toResponse $ "New count is: " ++ show c
  ]
```

Note that we do not call the `incCountBy` and `peekCount` functions directly. Instead we invoke them using the `update'` and `query'` functions:

```
update' :: (UpdateEvent event, MonadIO m) =>
           AcidState (EventState event) -- ^ handle to acid-state
        -> event                        -- ^ update event to execute
        -> m (EventResult event)
query'  :: (QueryEvent event , MonadIO m) =>
           AcidState (EventState event) -- ^ handle to acid-state
        -> event                        -- ^ query event to execute
        -> m (EventResult event)
```

Thanks to `makeAcidic`, the functions that we originally defined now have types with the same name, but starting with an uppercase letter:

```
data PeekCount  = PeekCount
data IncCountBy = IncCountBy Integer
```

The arguments to the constructors are the same as the arguments to the original function.

So now we can decipher the meaning of the type for the `update'` and `query'` functions. For example, in this code:

```
c <- update' acid (IncCountBy 1)
```

The event is `(IncCountBy 1)` which has the type `IncCountBy`. Since there is an `UpdateEvent IncCountBy` instance, we can use this event with the `update'` function. That gives us:

```
update' :: (UpdateEvent IncCountBy, MonadIO m) =>
           AcidState (EventState IncCountBy)
        -> IncCountBy
        -> m (EventResult IncCountBy)
```

`EventState` is a type function. `EventState IncCountBy` results in the type `CounterState`. So that reduces to `AcidState CounterState`. So, we see that we can not accidently call the `IncCountBy` event against an acid state handle of the wrong type.

`EventResult` is also a type function. `EventResult IncCountBy` is `Integer`, as we would expect from the type signature for `IncCountBy`.

As mentioned earlier, the underlying update and query events we created are pure functions. But, in order to have a durable database (aka, be able to recover after powerloss, etc) we do need to log the update events to disk so that we can reply them in the event of a recovery. So, rather than invoke our update and query events directly, we call them indirectly via the `update` and `query` functions. `update` and `query` interact with the `acid-state` system to ensure that the acid-state events are properly logged, called in the correct order, run atomitically and isolated, etc.

There is no way in Haskell to save a function to disk or send it over the network. So, `acid-state` has to cheat a little. Instead of storing the function, it just stores the name of the function and the value of its arguments. That is what the `IncCountBy` type is for – it is the value that can be serialized and saved to disk or sent over the network.

Finally, we have our main function:

```
main :: IO ()
main =
  bracket (openLocalState initialCounterState)
          (createCheckpointAndClose)
           (\acid ->
               simpleHTTP nullConf (handlers acid))
```

`openLocalState` starts up `acid-state` and returns an handle. If existing state is found on the disk, it will be automatically restored and used. If no pre-existing state is found, then `initialCounterState` will be used. `openLocalState` stores data in a directory named `state/[typeOf state]`. In this example, that would be, `state/CounterState`. If you want control over where the state information is stored use `openLocalStateFrom` instead.

The shutdown sequence creates a checkpoint when the server exits. This is good practice because it helps the server start faster, and makes migration go more smoothly. Calling `createCheckpointAndClose` is not critical to data integrity. If the server crashes unexpectedly, it will replay all the logged transactions (Durability). However, it is a good idea to create a checkpoint on close. If you change an existing update event, and then tried to replay old versions of the event, things would probably end poorly. However, restoring from a checkpoint does not require the old events to be replayed. Hence, always creating a checkpoint on shutdown makes it easier to upgrade the server.

Source code for the app is here.

IxSet: a set with multiple indexed keys

To use `IxSet` you will need to install the optional `ixset` package.

In the first `acid-state` example we stored a single value. But in real database we typically need to store a large collection of records. And we want to be able to efficiently search and update those records. For simple key/value pairs we can use `Data.Map`. However, in practice, we often want to have *multiple* keys. That is what `IxSet` set offers – a set-like type which can be indexed by multiple keys.

Instead of having:

```
Set Foo
```

we will have:

```
IxSet Foo
```

with the ability to do queries based on the indices of `Foo`, which are defined using the `Indexable` type-class.

IxSet can be found here on hackage.

In this example, we will use `IxSet` to create a mini-blog.

```
{-# LANGUAGE DeriveDataTypeable, GeneralizedNewtypeDeriving,
  RecordWildCards, TemplateHaskell, TypeFamilies,
  OverloadedStrings #-}

module Main where

import Control.Applicative  ((<$>), optional)
import Control.Exception    (bracket)
import Control.Monad        (msum, mzero)
import Control.Monad.Reader (ask)
import Control.Monad.State  (get, put)
import Control.Monad.Trans  (liftIO)
import Data.Acid            ( AcidState, Update, Query
                            , makeAcidic, openLocalState
                            )
import Data.Acid.Advanced   (update', query')
import Data.Acid.Local      (createCheckpointAndClose)
import Data.Data            (Data, Typeable)
import Data.IxSet           ( Indexable(..), IxSet(..), (@=)
                            , Proxy(..), getOne, ixFun, ixSet )
import qualified Data.IxSet as IxSet
import Data.SafeCopy        (SafeCopy, base, deriveSafeCopy)
import Data.Text            (Text)
import Data.Text.Lazy       (toStrict)
import qualified Data.Text  as Text
import Data.Time            (UTCTime(..), getCurrentTime)
```

```
import Happstack.Server
    ( ServerPart, Method(POST, HEAD, GET), Response, decodeBody
    , defaultBodyPolicy, dir, lookRead, lookText, method
    , notFound, nullConf, nullDir, ok, seeOther, simpleHTTP
    , toResponse)
import           Text.Blaze.Html ((!), Html)
import qualified Text.Blaze.Html4.Strict as H
import qualified Text.Blaze.Html4.Strict.Attributes as A
```

The first thing we are going to need is a type to represent a blog post.

It is convenient to assign a unique id to each blog post so that it can be easily referenced in URLs and easily queried in the `IxSet`. In order to keep ourselves sane, we can create a `newtype` wrapper around an `Integer` instead of just using a nameless `Integer`.

```
newtype PostId = PostId { unPostId :: Integer }
    deriving (Eq, Ord, Data, Enum, Typeable)

$(deriveSafeCopy 0 'base ''PostId)
```

Note that in addition to deriving normal classes like `Eq` and `Ord`, we use template haskell to derive an instance of `SafeCopy`. This is not required by `IxSet` itself, but since we want to store the our blog posts in `acid-state` we will need it there.

A blog post will be able to have two statuses 'draft' and 'published'. We could use a boolean value, but it is easier to understand what `Draft` and `Published` mean instead of trying to remember what `True` and `False` mean. Additionally, we can easily extend the type with additional statuses later.

```
data Status =
    Draft
  | Published
    deriving (Eq, Ord, Data, Typeable)

$(deriveSafeCopy 0 'base ''Status)
```

And now we can create a simple record which represents a single blog post:

```
data Post = Post
    { postId  :: PostId
    , title   :: Text
    , author  :: Text
    , body    :: Text
    , date    :: UTCTime
    , status  :: Status
    , tags    :: [Text]
    }
    deriving (Eq, Ord, Data, Typeable)
```

```
$(deriveSafeCopy 0 'base ''Post)
```

Each `IxSet` key needs to have a unique type. Looking at `Post` it seems like that could be trouble – because we have multiple fields which all have the type `Text`. Fortunately, we can easily get around this by introducing some newtypes which are used for indexing.

```
newtype Title     = Title Text
    deriving (Eq, Ord, Data, Typeable)
$(deriveSafeCopy 0 'base ''Title)

newtype Author    = Author Text
    deriving (Eq, Ord, Data, Typeable)
$(deriveSafeCopy 0 'base ''Author)

newtype Tag       = Tag Text
    deriving (Eq, Ord, Data, Typeable)
$(deriveSafeCopy 0 'base ''Tag)

newtype WordCount = WordCount Int
    deriving (Eq, Ord, Data, Typeable)
$(deriveSafeCopy 0 'base ''WordCount)
```

Defining the indexing keys

We are now ready to create an instance of the `Indexable` class. This is the class that defines the keys for a `Post` so that we can store it in an `IxSet`:

```
instance Indexable Post where
  empty = ixSet
    [ ixFun $ \bp -> [ postId bp ]
    , ixFun $ \bp -> [ Title  $ title bp  ]
    , ixFun $ \bp -> [ Author $ author bp ]
    , ixFun $ \bp -> [ status bp ]
    , ixFun $ \bp -> map Tag (tags bp)
    , ixFun $ (:[]) . date  -- point-free, just for variety
    , ixFun $ \bp -> [ WordCount (length $ Text.words $ body bp) ]
    ]
```

In the `Indexable Post` instance we create a list of `Ix Post` values by using the `ixFun` helper function:

```
ixFun :: (Ord b, Typeable b) => (a -> [b]) -> Ix a
```

We pass a key extraction function to `ixFun`. For example, in this line:

```
ixFun $ \bp -> [ postId bp ]
```

we extract the `PostId` from a `Post`. Note that we return a list of keys values not just a single key. That is because a single entry might have several keys for

a specific type. For example, a `Post` has a list of tags. But, we want to be able to search for posts that match a specific tag. So, we index each tag separately:

```
ixFun $ \bp -> map Tag (tags bp)
```

Note that the keys do not have to directly correspond to a field in the record. We can perform calculations to create arbitrary keys. For example, the `WordCount` key calculates the number of words in a post:

```
ixFun $ \bp -> [ WordCount (length $ Text.words $ body bp) ]
```

For the `Title` and `Author` keys we add the newtype wrapper.

Now we will create the record that we will use with acid-state to hold the `IxSet Post` and other state information.

```
data Blog = Blog
    { nextPostId :: PostId
    , posts      :: IxSet Post
    }
    deriving (Data, Typeable)

$(deriveSafeCopy 0 'base ''Blog)

initialBlogState :: Blog
initialBlogState =
    Blog { nextPostId = PostId 1
         , posts      = empty
         }
```

`IxSet` does not (currently) provide any auto-increment functionality for indexes, so we have to keep track of what the next available `PostId` is ourselves. That is why we have the `nextPostId` field. (Feel free to submit a patch that adds an auto-increment feature to `IxSet`!).

Note that in `initialBlogState` the `nextPostId` is initialized to 1 not 0. Sometimes we want to create a `Post` that is not yet in the database, and hence does not have a valid `PostId`. I like to reserve `PostId 0` to mean uninitialized. If I ever see a `PostId 0` stored in the database, I know there is a bug in my code.

Inserting a Record

Next we will create some update and query functions for our acid-state database.

```
-- | create a new, empty post and add it to the database
newPost :: UTCTime -> Update Blog Post
newPost pubDate =
    do b@Blog{..} <- get
       let post = Post { postId = nextPostId
                       , title  = Text.empty
                       , author = Text.empty
```

```
                    , body   = Text.empty
                    , date   = pubDate
                    , status = Draft
                    , tags   = []
                    }
      put $ b { nextPostId = succ nextPostId
              , posts      = IxSet.insert post posts
              }
      return post
```

Nothing in that function should be too surprising. We have to pass in `UTCTime`, because we can not do IO in the update function. Because `PostId` is an instance of `Enum` we can use `succ` to increment it. To add the new post to the `IxSet` we use `IxSet.insert`.

```
insert :: (Typeable a, Ord a, Indexable a) =>
          a -> IxSet a -> IxSet a
```

Updating a Record

Next we have a function that updates an existing `Post` in the database with a newer version:

```
-- | update the post in the database (indexed by PostId)
updatePost :: Post -> Update Blog ()
updatePost updatedPost = do
  b@Blog{..} <- get
  put $ b { posts =
              IxSet.updateIx (postId updatedPost) updatedPost posts
          }
```

Note that instead of `insert` we use `updateIx`:

```
updateIx :: (Indexable a, Ord a, Typeable a, Typeable key) =>
            key
         -> a
         -> IxSet a
         -> IxSet a
```

The first argument to `updateIx` is a key that maps to the post we want to updated in the database. The key must uniquely identify a single entry in the database. In this case we use our primary key, `PostId`.

Looking up a value by its indexed key

Next we have some query functions.

```
postById :: PostId -> Query Blog (Maybe Post)
postById pid =
     do Blog{..} <- ask
        return $ getOne $ posts @= pid
```

`postById` is used to lookup a specific post by its `PostId`. This is our first example of querying an `IxSet`. Here we use the equals query operator:

```
(@=) :: (Typeable key, Ord a, Typeable a, Indexable a) =>
        IxSet a -> key -> IxSet a
```

It takes an `IxSet` and filters it to produce a new `IxSet` which only contains values that match the specified key. In this case, we have specified the primary key (`PostId`), so we expect exactly zero or one values in the resulting `IxSet`. We can use `getOne` to turn the result into a simple `Maybe` value:

```
getOne :: Ord a => IxSet a -> Maybe a
```

Ordering the Results and the `Proxy` type

Here is a query function that gets all the posts with a specific status (`Published` vs `Draft`) and sorts them in reverse chronological order (aka, newest first):

```
postsByStatus :: Status -> Query Blog [Post]
postsByStatus status = do
 Blog{..} <- ask
 let posts' =
       IxSet.toDescList (Proxy :: Proxy UTCTime) $
         posts @= status
 return posts'
```

We use the `@=` operator again to select just the posts which have the matching status. Since the publication date is a key (`UTCTime`) we can use `toDescList` to return a sorted list:

```
toDescList :: (Typeable k, Typeable a, Indexable a) =>
              Proxy k -> IxSet a -> [a]
```

`toDescList` takes a funny argument `(Proxy :: Proxy UTCTime)`. While the `Post` type itself has an `Ord` instance – we generally want to order by a specific key, which may have a different ordering. Since our keys are specified by type, we need a way to pass a type to `toDescList` so that it knows which key we want to order by. The `Proxy` type exists for that sole reason:

```
data Proxy a = Proxy
```

It just gives us a place to stick a type signature that `toDescList` and other functions can use.

Summary

You have now seen the basics of using `IxSet`. `IxSet` includes numerous other operations such as range-based queries, deleting records, converting to and from lists and Sets. See the haddock docs for a complete list of functions and their descriptions. You should have no difficulty understanding what they do based on what we have already seen.

Rest of the Example Code

The remainder of the code in this section integrates the above code into a fully functioning example. In order to keep things simple I have just used `blaze-html`. In a real application I would use `reform` to deal with the form generation and validation. (I would probably also use `web-routes` to provide type-safe URLs, and `HSP` for the templates). But those topics are covered elsewhere. The remainder of the code in this section does not contain any new concepts that have not already been covered in previous sections of the crash course.

```
$(makeAcidic ''Blog
  [ 'newPost
  , 'updatePost
  , 'postById
  , 'postsByStatus
  ])

-- | HTML template that we use to render all the
--   pages on the site
template :: Text -> [Html] -> Html -> Response
template title headers body =
  toResponse $
    H.html $ do
      H.head $ do
        css
        H.title (H.toHtml title)
        H.meta ! A.httpEquiv "Content-Type"
               ! A.content "text/html;charset=utf-8"
        sequence_ headers
      H.body $ do
        H.ul ! A.id "menu" $ do
         H.li $ H.a ! A.href "/" $ "home"
         H.li $ H.a ! A.href "/drafts" $ "drafts"
         H.li $ H.form ! A.enctype "multipart/form-data"
                       ! A.method "POST"
                       ! A.action "/new" $ H.button $ "new post"
        body

-- | CSS for our site
--
-- Normally this would live in an external .css file.
-- It is included inline here to keep the example
-- self-contained.
css :: Html
css =
 let s = Text.concat
      [ "body { color: #555; padding: 0; margin: 0; margin-left: 1em;}"
      , "ul { list-style-type: none; }"
      , "ol { list-style-type: none; }"
```

```
      , "h1 { font-size: 1.5em; color: #555; margin: 0; }"
      , ".author { color: #aaa; }"
      , ".date { color: #aaa; }"
      , ".tags { color: #aaa; }"
      , ".post { border-bottom: 1px dotted #aaa; margin-top: 1em; }"
      , ".bdy  { color: #555; margin-top: 1em; }"
      , ".post-footer { margin-top: 1em; margin-bottom: 1em; }"
      , "label { display: inline-block; width: 3em; }"
      , "#menu { margin: 0; padding: 0; margin-left: -1em;"
      ,         "border-bottom: 1px solid #aaa; }"
      , "#menu li { display: inline; margin-left: 1em; }"
      , "#menu form { display: inline; margin-left: 1em; }"
      ]
 in H.style ! A.type_ "text/css" $ H.toHtml s

-- | edit an existing blog post
edit :: AcidState Blog -> ServerPart Response
edit acid = do
 pid   <- PostId <$> lookRead "id"
 mMsg  <- optional $ lookText "msg"
 mPost <- query' acid (PostById pid)
 case mPost of
  Nothing ->
   notFound $ template "no such post" [] $
                do "Could not find a post with id "
                   H.toHtml (unPostId pid)
  (Just p@(Post{..})) -> msum
    [ do method GET
         ok $ template "foo" [] $ do
         case mMsg of
           (Just msg) | msg == "saved" -> "Changes saved!"
           _ -> ""
         H.form ! A.enctype "multipart/form-data"
                ! A.method "POST"
                ! A.action (H.toValue $ "/edit?id=" ++
                                        (show $ unPostId pid)) $ do
           H.label "title" ! A.for "title"
           H.input ! A.type_ "text"
                   ! A.name "title"
                   ! A.id "title"
                   ! A.size "80"
                   ! A.value (H.toValue title)
           H.br
           H.label "author" ! A.for "author"
           H.input ! A.type_ "text"
                   ! A.name "author"
```

```
                ! A.id "author"
                ! A.size "40"
                ! A.value (H.toValue author)
          H.br
          H.label "tags" ! A.for "tags"
          H.input ! A.type_ "text"
                  ! A.name "tags"
                  ! A.id "tags"
                  ! A.size "40"
                  ! A.value (H.toValue $
                              Text.intercalate ", " tags)
          H.br
          H.label "body" ! A.for "body"
          H.br
          H.textarea ! A.cols "80"
                     ! A.rows "20"
                     ! A.name "body" $ H.toHtml body
          H.br
          H.button ! A.name "status"
                   ! A.value "publish" $ "publish"
          H.button ! A.name "status"
                   ! A.value "save"    $ "save as draft"
    , do method POST
         ttl   <- lookText' "title"
         athr  <- lookText' "author"
         tgs   <- lookText' "tags"

         bdy   <- lookText' "body"
         now   <- liftIO $ getCurrentTime
         stts  <- do s <- lookText' "status"
                     case s of
                        "save"    -> return Draft
                        "publish" -> return Published
                        _         -> mzero
         let updatedPost =
                 p { title  = ttl
                   , author = athr
                   , body   = bdy
                   , date   = now
                   , status = stts
                   , tags   =
                       map Text.strip $ Text.splitOn "," tgs
                   }
         update' acid (UpdatePost updatedPost)
         case status of
           Published ->
```

```
            seeOther ("/view?id=" ++ (show $ unPostId pid))
                     (toResponse ())
          Draft     ->
            seeOther ("/edit?msg=saved&id=" ++
                      (show $ unPostId pid))
                     (toResponse ())
                ]

      where lookText' = fmap toStrict . lookText

-- | create a new blog post in the database,
--   and then redirect to /edit
new :: AcidState Blog -> ServerPart Response
new acid = do
  method POST
  now <- liftIO $ getCurrentTime
  post <- update' acid (NewPost now)
  let url = "/edit?id=" ++ show (unPostId $ postId post)
  seeOther url (toResponse ())

-- | render a single blog post into an HTML fragment
postHtml  :: Post -> Html
postHtml (Post{..}) =
  H.div ! A.class_ "post" $ do
    H.h1 $ H.toHtml title
    H.div ! A.class_ "author" $
       do "author: "
          H.toHtml author
    H.div ! A.class_ "date"   $
       do "published: "
          H.toHtml (show date)
    H.div ! A.class_ "tags"   $
       do "tags: "
          H.toHtml (Text.intercalate ", " tags)
    H.div ! A.class_ "bdy" $ H.toHtml body
    H.div ! A.class_ "post-footer" $ do
     H.span $ H.a !
       A.href (H.toValue $ "/view?id=" ++
                 show (unPostId postId)) $ "permalink"
     H.span $ " "
     H.span $ H.a !
       A.href (H.toValue $ "/edit?id=" ++
                 show (unPostId postId)) $ "edit this post"

-- | view a single blog post
view :: AcidState Blog -> ServerPart Response
view acid =
```

```
    do pid <- PostId <$> lookRead "id"
       mPost <- query' acid (PostById pid)
       case mPost of
         Nothing ->
             notFound $ template "no such post" [] $
               do "Could not find a post with id "
                  H.toHtml (unPostId pid)
         (Just p) ->
             ok $ template (title p) [] $ do
                 (postHtml p)

-- | render all the Published posts (ordered newest to oldest)
home :: AcidState Blog -> ServerPart Response
home acid =
    do published <- query' acid (PostsByStatus Published)
       ok $ template "home" [] $ do
         mapM_ postHtml published

-- | show a list of all unpublished blog posts
drafts :: AcidState Blog -> ServerPart Response
drafts acid =
    do drafts <- query' acid (PostsByStatus Draft)
       case drafts of
         [] -> ok $ template "drafts" [] $
               "You have no unpublished posts at this time."
         _ ->
             ok $ template "home" [] $
                 H.ol $ mapM_ editDraftLink drafts
 where
  editDraftLink Post{..} =
    let url = (H.toValue $ "/edit?id=" ++ show (unPostId postId))
    in H.a ! A.href url $ H.toHtml title

-- | route incoming requests
route :: AcidState Blog -> ServerPart Response
route acid =
    do decodeBody (defaultBodyPolicy "/tmp/" 0 1000000 1000000)
       msum [ dir "favicon.ico" $ notFound (toResponse ())
            , dir "edit"        $ edit acid
            , dir "new"         $ new acid
            , dir "view"        $ view acid
            , dir "drafts"      $ drafts acid
            , nullDir          >> home acid
            ]

-- | start acid-state and the http server
main :: IO ()
main =
```

```
    do bracket (openLocalState initialBlogState)
               (createCheckpointAndClose)
               (\acid ->
                    simpleHTTP nullConf (route acid))
```

Source code for the app is here.

Passing multiple `AcidState` handles around transparently

Manually passing around the `AcidState` handle gets tedious very quickly. A common solution is to stick the `AcidState` handle in a `ReaderT` monad. For example:

```
newtype MyApp = MApp {
 unMyApp :: ReaderT (AcidState AppState) (ServerPartT IO) Response
 }
```

We could then write some variants of the `update` and `query` functions which automatically retrieve the acid handle from `ReaderT`.

In this section we will show a slightly more sophisticated version of that solution which allows us to work with multiple `AcidState` handles and works well even if our app can be extended with optional plugins that contain additional `AcidState` handles.

```
{-# LANGUAGE DeriveDataTypeable, FlexibleContexts
  , GeneralizedNewtypeDeriving, MultiParamTypeClasses
  , OverloadedStrings, ScopedTypeVariables, TemplateHaskell
  , TypeFamilies, FlexibleInstances #-}

module Main where

import Control.Applicative  (Applicative, Alternative, (<$>))
import Control.Exception.Lifted    (bracket)
import Control.Monad.Trans.Control (MonadBaseControl)
import Control.Monad        (MonadPlus, mplus)
import Control.Monad.Reader ( MonadReader, ReaderT(..)
                            , ask)
import Control.Monad.State  (get, put)
import Control.Monad.Trans  (MonadIO(..))
import Data.Acid
    ( AcidState(..), EventState(..), EventResult(..)
    , Query(..), QueryEvent(..), Update(..), UpdateEvent(..)
    , IsAcidic(..), makeAcidic, openLocalState
    )
import Data.Acid.Local      ( createCheckpointAndClose
                            , openLocalStateFrom
```

```
                            )
import Data.Acid.Advanced   (query', update')
import Data.Maybe           (fromMaybe)
import Data.SafeCopy        (SafeCopy, base, deriveSafeCopy)
import Data.Data            (Data, Typeable)
import Data.Text.Lazy       (Text)
import Happstack.Server
    ( Happstack, HasRqData, Method(GET, POST), Request(rqMethod)
    , Response
    , ServerPartT(..), WebMonad, FilterMonad, ServerMonad
    , askRq, decodeBody, dir, defaultBodyPolicy, lookText
    , mapServerPartT, nullConf, nullDir, ok, simpleHTTP
    , toResponse
    )
import Prelude hiding       (head, id)
import System.FilePath      ((</>))
import Text.Blaze           ((!))
import Text.Blaze.Html4.Strict
    (body, head, html, input, form, label, p, title, toHtml)
import Text.Blaze.Html4.Strict.Attributes
    (action, enctype, for, id, method, name, type_, value)
```

The first thing we have is a very general class that allows us to retrieve a specific `AcidState` handle by its type from an arbitrary monad:

```
class HasAcidState m st where
   getAcidState :: m (AcidState st)
```

Next we redefine `query` and `update` so that they use `getAcidState` to automatically retrieve the the correct `AcidState` handle from whatever monad they are in:

```
query :: forall event m.
         ( Functor m
         , MonadIO m
         , QueryEvent event
         , HasAcidState m (EventState event)
         ) =>
         event
      -> m (EventResult event)
query event =
    do as <- getAcidState
       query' (as :: AcidState (EventState event)) event

update :: forall event m.
          ( Functor m
          , MonadIO m
          , UpdateEvent event
```

```
          , HasAcidState m (EventState event)
          ) =>
          event
       -> m (EventResult event)
update event =
    do as <- getAcidState
       update' (as :: AcidState (EventState event)) event

-- | bracket the opening and close of the 'AcidState' handle.

-- automatically creates a checkpoint on close
withLocalState
  :: ( MonadBaseControl IO m
     , MonadIO m
     , IsAcidic st
     , Typeable st
     , SafeCopy st
     ) =>
     Maybe FilePath         -- ^ path to state directory
  -> st                     -- ^ initial state value
  -> (AcidState st -> m a) -- ^ function which uses the
                            --   'AcidState' handle
  -> m a
withLocalState mPath initialState =
  bracket (liftIO $ open initialState)
          (liftIO . createCheckpointAndClose)
  where
    open = maybe openLocalState openLocalStateFrom mPath
```

(These functions will eventually reside in `acid-state` itself, or some other library).

Now we can declare a couple `acid-state` types:

```
-- State that stores a hit count

data CountState = CountState { count :: Integer }
    deriving (Eq, Ord, Data, Typeable, Show)

$(deriveSafeCopy 0 'base ''CountState)

initialCountState :: CountState
initialCountState = CountState { count = 0 }

incCount :: Update CountState Integer
incCount =
    do (CountState c) <- get
       let c' = succ c
```

```
      put (CountState c')
      return c'

$(makeAcidic ''CountState ['incCount])

-- State that stores a greeting

data GreetingState = GreetingState {  greeting :: Text }
               deriving (Eq, Ord, Data, Typeable, Show)

$(deriveSafeCopy 0 'base ''GreetingState)

initialGreetingState :: GreetingState
initialGreetingState = GreetingState { greeting = "Hello" }

getGreeting :: Query GreetingState Text
getGreeting = greeting <$> ask

setGreeting :: Text -> Update GreetingState ()
setGreeting txt = put $ GreetingState txt

$(makeAcidic ''GreetingState ['getGreeting, 'setGreeting])
```

Now that we have two states we can create a type to bundle them up like:

```
data Acid = Acid
   { acidCountState    :: AcidState CountState
   , acidGreetingState :: AcidState GreetingState
   }

withAcid :: Maybe FilePath -> (Acid -> IO a) -> IO a
withAcid mBasePath action =
  let basePath = fromMaybe "_state" mBasePath
      countPath = Just $ basePath </> "count"
      greetPath = Just $ basePath </> "greeting"
  in withLocalState countPath initialCountState    $ \c ->
     withLocalState greetPath initialGreetingState $ \g ->
      action (Acid c g)
```

Now we can create our `App` monad that stores the `Acid` in the `ReaderT`:

```
newtype App a = App { unApp :: ServerPartT (ReaderT Acid IO) a }
    deriving ( Functor, Alternative, Applicative, Monad
             , MonadPlus, MonadIO, HasRqData, ServerMonad
             , WebMonad Response, FilterMonad Response
             , Happstack, MonadReader Acid
             )

runApp :: Acid -> App a -> ServerPartT IO a
```

```
runApp acid (App sp) =
    mapServerPartT (flip runReaderT acid) sp
```

And finally, we need to write the `HasAcidState` instances:

```
instance HasAcidState App CountState where
    getAcidState = acidCountState    <$> ask

instance HasAcidState App GreetingState where
    getAcidState = acidGreetingState <$> ask
```

And that's it. We can now use `update` and `query` in the remainder of our code with out having to worry about the `AcidState` argument anymore.

Here is a page function that uses both the `AcidStates` in a transparent manner:

```
page :: App Response
page = do
  nullDir
  g <- greet
  c <- update IncCount -- ^ a CountState event
  ok $ toResponse $
     html $ do
       head $ do
         title "acid-state demo"
       body $ do
         form ! action "/"
              ! method "POST"
              ! enctype "multipart/form-data" $ do
           label "new message: " ! for "msg"
           input ! type_ "text" ! id "msg" ! name "greeting"
           input ! type_ "submit" ! value "update message"
         p $ toHtml g
         p $ do "This page has been loaded "
                toHtml c
                " time(s)."
  where
  greet = do
    m <- rqMethod <$> askRq
    case m of
      POST -> do
        decodeBody (defaultBodyPolicy "/tmp/" 0 1000 1000)
        newGreeting <- lookText "greeting"
        -- a GreetingState event
        update (SetGreeting newGreeting)
        return newGreeting
      GET  -> do
        -- a GreetingState event
```

```
        query GetGreeting
```

If have used `happstack-state` in the past, then this may remind you of how `happstack-state` worked. However, there is a critical different. In `happstack-state` it was possible to call `update` and `query` on events for state components that were not actually loaded. In this solution, however, the `HasAcidState` class ensures that we can only call `update` and `query` for valid `AcidState` handles.

Our main function is simply:

```
main :: IO ()
main =
    withAcid Nothing $ \acid ->
        simpleHTTP nullConf $ runApp acid page
```

Optional Plugins/Components

In an upcoming section we will explore various methods of extending your app via plugins and 3rd party libraries. These plugins and libraries may contain their own `AcidState` components. Very briefly, we will show how that might be handled.

Let's imagine we have this dummy plugin:

```
newtype FooState = FooState { foo :: Text }
    deriving (Eq, Ord, Data, Typeable)
$(deriveSafeCopy 0 'base ''FooState)

initialFooState :: FooState
initialFooState = FooState { foo = "foo" }

askFoo :: Query FooState Text
askFoo = foo <$> ask

$(makeAcidic ''FooState ['askFoo])

fooPlugin :: (Happstack m, HasAcidState m FooState) => m Response
fooPlugin =
    dir "foo" $ do
       txt <- query AskFoo
       ok $ toResponse txt
```

We could integrate it into our app by extending the `Acid` type to hold the `FooState` and then add an appropriate `HasAcidState` instance:

```
data Acid' = Acid'
    { acidCountState'    :: AcidState CountState
    , acidGreetingState' :: AcidState GreetingState
    , acidFooState'      :: AcidState FooState
    }
```

```
withAcid' :: Maybe FilePath -> (Acid' -> IO a) -> IO a
withAcid' mBasePath action =
  let basePath = fromMaybe "_state" mBasePath
      countPath = (Just $ basePath </> "count")
      greetPath = (Just $ basePath </> "greeting")
      fooPath   = (Just $ basePath </> "foo")
  in withLocalState countPath initialCountState    $ \c ->
     withLocalState greetPath initialGreetingState $ \g ->
     withLocalState fooPath   initialFooState      $ \f ->
       action (Acid' c g f)

newtype App' a = App'
    { unApp' :: ServerPartT (ReaderT Acid' IO) a
    }
    deriving ( Functor, Alternative, Applicative, Monad
             , MonadPlus, MonadIO, HasRqData, ServerMonad
             , WebMonad Response, FilterMonad Response
             , Happstack, MonadReader Acid'
             )

instance HasAcidState App' FooState where
    getAcidState = acidFooState' <$> ask
```

Now we can use `fooAppPlugin` like any other part in our app:

```
fooAppPlugin :: App' Response
fooAppPlugin = fooPlugin
```

An advantage of this method is that `fooPlugin` could also have access to the other `AcidState` components like `CountState` and `GreetingState`.

A different option would be for `fooPlugin` to use its own `ReaderT`

```
fooReaderPlugin
  :: ReaderT (AcidState FooState) (ServerPartT IO) Response
fooReaderPlugin = fooPlugin

instance HasAcidState
           (ReaderT (AcidState FooState) (ServerPartT IO))
           FooState where
    getAcidState = ask

withFooPlugin :: (MonadIO m, MonadBaseControl IO m) =>
     FilePath                           -- ^ path to state directory
  -> (ServerPartT IO Response -> m a)   -- ^ function that
                                        --   uses 'fooPlugin'
  -> m a
withFooPlugin basePath f =
  let fooPath = (Just $ basePath </> "foo") in
```

```
  withLocalState fooPath initialFooState $ \fooState ->
    f $ runReaderT fooReaderPlugin fooState

main' :: IO ()
main' =
  withFooPlugin "_state" $ \fooPlugin' ->
    withAcid Nothing $ \acid ->
      simpleHTTP nullConf $ fooPlugin' `mplus` runApp acid page
```

We will come back to this in detail later when we explore plugins and libraries.

Source code for the app is here.

Using Template Haskell

Template Haskell is a GHC extension that makes it possible to generate new code at compile time. It is like a more powerful version of C macros, but a bit more restrictive than LISP macros. You can see the code that is being generated by passing the `-ddump-splices` flag to GHC.

There are only a few places in Happstack where you will encounter Template Haskell code. In each of those cases, it is used to generate some very boiler-plate code. You are already familiar with one code generation mechanism in Haskell – the `deriving (Eq, Ord, Read, Show, Data, Typeable)` clause. In Happstack, we use Template Haskell in a similar manner to derive instances of classes like `SafeCopy` and `IsAcidic`.

There are only a few simple things you will need to learn to use Template Haskell with Happstack.

To enable Template Haskell you will need to include `{-# LANGUAGE TemplateHaskell #-}` at the top of the file.

Here is an example of some Template Haskell that derives a `SafeCopy` instance:

```
$(deriveSafeCopy 0 'base ''CounterState)
```

There are three new pieces of syntax you will see:

`$( )` This syntax is used to indicate that the code inside is going to generate code. The `$(..)` will be replaced by the generated code, and then the module will be compiled.

`'` The single quote in `'base` is syntax that returns the `Name` of a function or constructor. (Specificially, `Language.Haskell.TH.Syntax.Name`).

`''` Note: that is two single ticks `''` not a double-quote `"`. It serves the same purpose as `'` except that it is used to get the `Name` of a type instead of a function or constructor.

Finally, you may occasionally run into some staging restrictions. In a normal Haskell source file, it does not matter what order you declare things. You can use a type in a type signature, and then define the type later in the file. However, when using Template Haskell, you may occasionally find that you need to order

your types so that they are declared before they are used. If the compiler complains that it can't find a type that you have clearly defined in the file, try moving the declaration up higher.

That is everything you should need to know to use Template Haskell in Happstack. See the relevant section of the crash course for the details of calling specific Template Haskell functions such as `deriveSafeCopy`.