

# Vocabulink

Chris Forno (jekor)

May 3rd, 2009

## 1 Introduction

This is Vocabulink, the FastCGI process that handles all web requests for <http://www.vocabulink.com/>.

Vocabulink is essentially a multi-user application that operates via the web. It's structured like a standalone application inasmuch as it handles multiple requests in a multi-threaded process. Yet, it operates as a CGI program. It's designed with the assumption that it may be only 1 of many processes servicing requests and that it doesn't have exclusive access to resources such as a database.

The program is built with GHC 6.8.3 using options `-Wall -fglasgow-exts -threaded` and with `-package fastcgi`. I keep the build free from warnings at all times (which sometimes leads to a few oddities in the source). It has been tested on GNU/Linux.

### 1.1 Copyright Notice

Copyright 2008, 2009 Chris Forno

Vocabulink is free software: you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Vocabulink is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License along with Vocabulink. If not, see <http://www.gnu.org/licenses/>.

### 1.2 Architecture

Requests arrive via a webserver.<sup>1</sup> They are passed to the `vocabulink.fcgi` process (this program) on TCP port 10033 of the local loopback interface.

Upon receiving a request (connection), we immediately fork a new thread. In this thread, we establish a connection to a PostgreSQL server (for each request). We then examine the thread for an authentication cookie. If it exists and is valid, we consider the request to have originated from an authenticated member. We pack both the database handle and the authenticated member information into our "App" monad ([section 7](#)).

**module *Main* where**

---

<sup>1</sup>I'm currently using nginx on [www.vocabulink.com](http://www.vocabulink.com), but it should work with any server that supports FastCGI.

## 2 Our Modules

These are the Vocabulink modules we need from the toplevel. They are grouped primarily based on division of labor. The exception is the App module. The App module defines the App monad and must make use of both database and CGI functions. In order to limit cyclical dependencies, it's broken out into a separate module.

```
import Vocabulink.App
```

Each of these modules will be described in its own section.

```
import Vocabulink.Article
import Vocabulink.DB
import Vocabulink.CGI
import Vocabulink.Forum
import Vocabulink.Html hiding (method, options)
import Vocabulink.Link
import Vocabulink.Member
import Vocabulink.Review
import Vocabulink.Utils
```

## 3 Other Modules

Vocabulink makes use of a half dozen or so Haskell libraries not included with GHC. Even though we don't use them all in this module, I'll describe them here so that they'll be more familiar as they're introduced.

**Codec.Binary.UTF8.String** Vocabulink would be pretty useless without being able to handle the writing systems of other languages. We only make use of 2 functions provided by this library: *encodeString* and *decodeString*. *decodeString* takes a UTF-8 string—either from the webserver or from the database—and converts it into a Unicode string that can be used by Haskell natively. We use *encodeString* to go in the other direction. Whenever we write out a string to the database, the webserver, or a log file; it needs to be encoded to UTF-8. This is something that the type system does not (yet) handle for us, so we need to be careful to correctly encode and decode strings.

**Data.ConfigFile** We need to have some parameters configurable at runtime. This allows us to do things differently in test and production environments. It also allows us to publish the source to the program without exposing sensitive information.

**Data.Digest.OpenSSL.HMAC** The nano-hmac library is used for generating tokens for use in member authentication tokens.

**Database.HDBC** We make heavy use of PostgreSQL via HDBC, as Vocabulink is a data-driven application. HDBC takes most of the work out of converting between types when exchanging data with the database.

**Network.FastCGI** The FastCGI library provides a simple interface that's mostly compatible with the Network.CGI library. I've modified the library slightly so that it outputs using UTF-8 by default.<sup>2</sup>

---

<sup>2</sup>I haven't released my changes to Network.FastCGI as I suspect there might be a better way to make UTF-8 output more general (if it doesn't exist already). But I'd be happy to share the changes if you're interested.

**Network.Gravatar** The gravatar library is a simple and convenient way to generate links to gravatar images. It was a bit of a pleasant surprise, and a sign of Haskell's maturity, to find it.

**Network.Memcache** Interacting with memcached is very simple because it uses a simple text protocol. However, a library already exists, so we take advantage of it. As with `Network.FastCGI`, I slightly modified this one to use UTF-8 output by default and to support the flush command.

**Network.URI** Various parts of the code may need to construct or deconstruct URLs. Using this library should be safer than using various string-mangling techniques throughout the code.

**Text.Formlets** Formlets are one of the unique advantages that we get from working in a functional language. The Formlets library isn't perfect yet, namely with the way field names are automatically generated, but it's useful regardless.

**Text.JSON** We make use of the JSON encoding to sending data to web browsers for AJAX-style interaction.

**Text.ParserCombinators.Parsec** We need to parse text from time to time. The dispatcher, the member authentication routines, and the article publishing system all make use of Parsec; and probably more will in the future.

**Text.Pandoc** Mnemonic stories and forum posts are handled by Pandoc using the Markdown formatting syntax. The text is stored in Markdown syntax in the database to avoid lossiness and is rendered by Pandoc upon retrieval. Note that Pandoc is not responsible for formatting articles though (those are handled by Muse Mode).

```
import Control.Concurrent (forkIO)
import Control.Monad (join)
import Control.Monad.Error (runErrorT)
import Data.ConfigFile (readfile, emptyCP, ConfigParser, CPError, options)
import Data.List (find, intercalate, intersect)
import Data.List.Split (splitOn)
import Network.FastCGI (runFastCGIConcurrent')
import Network.URI (URI (.), unEscapeString)
```

## 4 Entry and Dispatch

When the program starts, it immediately begin listening for connections. `runFastCGIConcurrent'` spawns up to 2,048 threads. This matches the number that nginx, running in front of `vocabulink.cgi`, is configured for. `handleErrors'` and `runApp` will be explained later. They basically catch unhandled database errors and pack information into the App monad.

Before forking, we read a configuration file. We pass this to `runApp` so that all threads have access to global configuration information.

The first thing we do after forking is establish a database connection. The database connection might be used immediately in order to log errors. It'll eventually be passed to the App monad where it'll be packed into a reader environment.

```

main :: IO ()
main = do cp' ← getConfig
      case cp' of
        Left e   → print e
        Right cp → runFastCGIConcurrent' forkIO 2048 (do
          c ← liftIO connect
          handleErrors' c (runApp c cp handleRequest))

```

The path to the configuration file is the one bit of configuration that's the same in all environments.

```

configFile :: String
configFile = "/etc/vocabulink.conf"

```

These config vars are required by the program in order to do anything useful. They are guaranteed to exist later and can safely be read with `forceEither $ get`.

```

requiredConfigVars :: [String]
requiredConfigVars = ["authtokensalt", "articledir", "staticdir",
                     "supportaddress"]

```

This retrieves the config file and makes sure that it contains all of the required configuration variables. We check the variables now because we want to find out about missing ones at program start time rather than in the logs later.

```

getConfig :: IO (Either CPErrors ConfigParser)
getConfig = runErrorT $ do
  cp ← join $ liftIO $ readfile emptyCP configFile
  opts ← options cp "DEFAULT"
  if requiredConfigVars `intersect` opts == requiredConfigVars
  then return cp
  else error "Missing configuration options."

```

`handleRequest` “digests” the requested URI before passing it to the dispatcher.

```

handleRequest :: App CGIResult
handleRequest = do
  uri ← requestURI
  method ← requestMethod
  let path = pathList uri
      dispatch' method path

```

We extract the path part of the URI, “unescape it” (convert characters), decode it (convert UTF-8 characters to Unicode Chars), and finally parse it into directory and filename components. For example,

```

/some/directory/and/a/filename

```

becomes

```

["some", "directory", "and", "a", "filename"]

```

Note that the parser does not have to deal with query strings or fragments because *uriPath* has already stripped them.

The one case this doesn't handle correctly is `//something`, because it's handled differently by *Network.CGI*.

```
pathList :: URI → [String]
pathList = splitOn "/" ∘ decodeString ∘ unEscapeString ∘ uriPath
```

Before we actually dispatch the request, we use the opportunity to clean up the URI and redirect the client if necessary. This handles cases like trailing slashes. We want only one URI to point to a resource.<sup>3</sup>

```
dispatch' :: String → [String] → App CGIResult
dispatch' method path =
  case path of
    [ "", "" ] → frontPage -- "/"
    ("":xs) → case find (≡ "") xs of
      Nothing → dispatch method xs
      Just _ → redirect $ "/" ++ (intercalate "/" $ filter (≠ "") xs)
    _ → output404 path
```

Here is where we dispatch each request to a function. We can match the request on method and path components. This means that we can dispatch a GET request to one function and a POST request to another.

```
dispatch :: String → [String] → App CGIResult
```

## 4.1 Articles

Some permanent URIs are essentially static files. To display them, we make use of the article system (formatting, metadata, etc). You could call these elevated articles. We use articles because the system for managing them exists already (revision control, etc)

Each `.html` file is actually an HTML fragment. These happen to be generated from Muse Mode files by Emacs, but we don't really care where they come from.

```
dispatch "GET" ["help"]           = articlePage "help"
dispatch "GET" ["privacy"]        = articlePage "privacy"
dispatch "GET" ["terms-of-use"]   = articlePage "terms-of-use"
dispatch "GET" ["source"]         = articlePage "source"
```

Other articles are dynamic and can be created without recompilation. We just have to rescan the filesystem for them. They also live in the `/article` namespace (specifically at `/article/title`).

```
dispatch "GET" ["article", x] = articlePage x
```

We have 1 page for getting a listing of all published articles.

```
dispatch "GET" ["articles"] = articlesPage
```

And this is a method used by the web-based administrative interface to reload the articles from the filesystem. (Articles are transmitted to the server via rsync using the filesystem, not through the web.)

```
dispatch "POST" ["articles"] = refreshArticles
```

---

<sup>3</sup>I'm not sure that this is the right thing to do. Would it be better just to give the client a 404?

## 4.2 Link Pages

Vocabulink revolves around links—the associations between words or ideas. As with articles, we have different functions for retrieving a single link or a listing of links. However, the dispatching is complicated by the fact that members can operate upon links (we need to handle the POST method).

If we could rely on the DELETE method being supported by all browsers, this would be a little less ugly. However, I've decided to only use GET and POST. All other methods are appended as an extra path component (here, as *method'*).<sup>4</sup>

For clarity, this dispatches:

```
GET /link/10           → linkPage
GET /link/something   → not found
GET /link/10/something → not found
POST /link/10/delete   → deleteLink
```

```
dispatch method path@("link" : x : method') = do
  case maybeRead x of
    Nothing → output404 path
    Just n  → case (method, method') of
      ("GET" , [])      → linkPage n
      ("POST", ["delete"]) → deleteLink n
      ( _      , -)      → output404 path
```

## 4.3 Searching

Retrieving a listing of links is easier.

Searching means forms and forms mean query strings. So if there's a `contains` in the query string for the links page, it will do a search. E.g.

```
GET /links?contains=water
```

```
dispatch "GET" ["links"] = do
  contains ← getInput "contains"
  maybe (linksPage "Latest Links" latestLinks) linksContainingPage contains
dispatch "GET" path@[ "links", x ] = do
  case maybeRead x of
    Nothing → output404 path
    Just n  → do
      memberName ← getMemberName n
      case memberName of
        Nothing → output404 path
        Just n'  → linksPage ("Links by " ++ n') (memberLinks n)
```

Creating a new link is a 2-step process. First, the member requests a page on which to enter information about the link. Then they POST the details to establish the link. (Previewing is done through the GET as well.)

```
dispatch "GET" ["link"] = newLink
dispatch "POST" ["link"] = newLink
```

---

<sup>4</sup>I'm not 100% satisfied with this design decision, but I haven't thought of a better way yet.

## 4.4 Link Review

Members review their links by interacting with the site in a vaguely REST-ish way. The intent behind this is that in the future they will be able to review their links through different means such as a desktop program or a phone application.

Because of the use of *withRequiredMemberNumber*, a logged out member will be redirected to a login page when attempting to review.

```
retrieve the next link for review → GET /review/next
mark link as reviewed           → POST /review/n
add a link for review           → POST /review/n/add
```

(where *n* is the link number)

```
dispatch method path@("review" : rpath) =
  withRequiredMemberNumber $ λmemberNo →
  case (method, rpath) of
    ("GET" , ["next"]) → nextReview memberNo
    ("POST", (x : xs)) → do
      case maybeRead x of
        Nothing → outputError 400
          "Links are identified by numbers only." []
        Just n → case xs of
          ["add"] → newReview memberNo n
          []      → linkReviewed memberNo n
          -       → output404 path
    (-      , -) → output404 path
```

## 4.5 Membership

Becoming a member is simply a matter of filling out a form.

```
dispatch "GET" ["member", "signup"] = registerMember
dispatch "POST" ["member", "signup"] = registerMember
```

But to use most of the site, we require email confirmation.

```
dispatch "GET" ["member", "confirmation"] = confirmEmailPage
dispatch "GET" ["member", "confirmation", x] = confirmEmail x
```

Logging in is a similar process.

```
dispatch "GET" ["member", "login"] = login
dispatch "POST" ["member", "login"] = login
```

Logging out can be done without a form.

```
dispatch "POST" ["member", "logout"] = logout
```

Members can also request support, if for some reason they can't or don't want to use the forums.

```
dispatch "GET" ["member", "support"] = memberSupport
dispatch "POST" ["member", "support"] = memberSupport
```

## 4.6 Forums

While Vocabulink is still growing (and into the future), it's important to help new members along and to get feedback from them. For this, Vocabulink uses forums.

You may begin to notice a dispatching pattern by now.

```
dispatch "GET" ["forums"] = forumsPage
dispatch "POST" ["forums"] = forumsPage
```

Forums are uniquely identified by their name. The names are trusted to be unique and reversibly mappable into URI-safe strings because they are created by administrators of the site.

```
dispatch "POST" ["forum", "new"] = createForum
dispatch "GET" ["forum", x] = forumPage x
```

However, topics can be created by anyone and are identified by numbers. This might seem like a lost opportunity for search engine optimization, but including the forum topic text could lead to some very long URIs.

```
dispatch "GET" ["forum", x, "new"] = newTopicPage x
dispatch "POST" ["forum", x, "new"] = newTopicPage x
```

```
dispatch "GET" path@[ "forum", x, y ] =
  case maybeRead y of
    Nothing → output404 path
    Just n   → forumTopicPage x n
```

“reply” and “preview” are used here as nouns.

```
dispatch "GET" ["comment", "reply"] = replyToForumComment
dispatch "POST" ["comment", "reply"] = replyToForumComment
```

```
dispatch "GET" ["comment", "preview"] = commentPreview
```

## 4.7 Everything Else

For Google Webmaster Tools, we need to respond to a certain URI that acts as a kind of “yes, we really do run this site”.

```
dispatch "GET" ["google46b9909165f12901.html"] = output' ""
```

It would be nice to automatically respond with “Method Not Allowed” on URIs that exist but don't make sense for the requested method (presumably POST). However, we need to take a simpler approach because of how the dispatch method was designed (pattern matching is limited). We output a qualified 404 error.

```
dispatch _ path = output404 path
```

Finally, we get to an actual page of the site: the front page. Currently, it's doing a lot more than I'd like it to do. But it'll have to stay this way until we have some sort of widget/layout

system. It gets the common header, footer, and associated functionality by using the *stdPage* function.

Logged-in members are presented with a different “article” in the main body as well as a “My Links” box showing them the links that they’ve created. The page also shows a list of recent articles should the reader feel a little lost or curious.

```
frontPage :: App CGIResult
frontPage = do
  memberNo ← asks appMemberNo
  my ← maybe (return noHtml) (myLinks) memberNo
  latest ← newLinks
  articles ← latestArticles
  let article = isJust memberNo ? "welcome-member" $ "welcome"
      article' ← getArticle article
      body ← maybe (return $ h1 << "Welcome to Vocabulink") articleBody article'
  stdPage "Welcome to Vocabulink" [JS "MochiKit", JS "page"] [] [
    thediv ! [identifier "main-content"] << body,
    thediv ! [identifier "sidebar"] << [
      latest, my, articles]]
  where myLinks mn = do
    ls ← memberLinks mn 0 10
    return $ maybe (noHtml) (λl → thediv ! [theclass "sidebar"] << [
      h3 << anchor ! [href ("/links/" ++ (show mn))] <<
        "My Links",
      unordList (map partialLinkHtml l) !
        [theclass "links"]]) ls
  newLinks = do
    ls ← latestLinks 0 10
    return $ maybe (noHtml) (λl → thediv ! [theclass "sidebar"] << [
      h3 << anchor ! [href "/links"] <<
        "Latest Links",
      unordList (map partialLinkHtml l) !
        [theclass "links"]]) ls
  latestArticles = do
    ls ← getArticles
    return $ maybe (noHtml) (λl → thediv ! [theclass "sidebar"] << [
      h3 << anchor ! [href "/articles"] <<
        "Latest Articles",
      unordList (map articleLinkHtml l)]) ls
```

## 5 Utility Functions

Here are some functions that aren’t specific to Vocabulink, but that don’t exist in any libraries I know of. We also use this module to export some oft-used functions for other modules.

```
module Vocabulink.Utils (
  if', (?), safeHead, currentDay, currentYear,
  formatSimpleTime, basename, translate, (< $$ >),
  sendMail,
  memcacheSet, memcacheDelete, memcacheFlush,
```

```

{-Codec.Binary.UTF8.String -}  encodeString, decodeString,
{-Control.Applicative -}      pure, (< $ >), (< * >),
{-Control.Applicative.Error -} Failing (.), maybeRead,
{-Control.Monad -}            liftM,
{-Control.Monad.Trans -}      liftIO, MonadIO,
{-Data.Char -}                 toLower,
{-Data.Maybe -}                maybe, fromMaybe, fromJust, isJust, catMaybes,
{-Data.Time.Calendar -}       Day,
{-Data.Time.Clock -}          UTCTime,
{-Data.Time.Format -}         formatTime,
{-Data.Time.LocalTime -}      ZonedTime,
{-System.Locale -}            defaultTimeLocale, rfc822DateFormat) where

```

Vocabulink deals with all strings as UTF8. Every part of the website potentially makes use of foreign writing systems. Occasionally I also like using other Unicode characters.

```
import Codec.Binary.UTF8.String (encodeString, decodeString)
```

Applicative was first introduced to Vocabulink for working with Formlets. However, it seems to be a style that the Haskell community is using more and more.

```
import Control.Applicative (pure, (< $ >), (< * >))
import Control.Applicative.Error (Failing (.), maybeRead)
import Control.Exception (try, bracket)
```

We make particularly extensive use of *liftM* and the Maybe monad.

```
import Control.Monad (liftM)
import Control.Monad.Trans (liftIO, MonadIO)
import Data.Char (toLower)
import Data.Maybe (maybe, fromMaybe, fromJust, isJust, catMaybes)
```

Time is notoriously difficult to deal with in Haskell. It gets especially tricky when working with the database and libraries that expect different formats.

```
import Data.Time.Calendar (Day, toGregorian)
import Data.Time.Clock (getCurrentTime, UTCTime)
import Data.Time.Format (formatTime)
import Data.Time.LocalTime (getCurrentTimeZone, utcToLocalTime,
                             LocalTime (.), ZonedTime)
import qualified Network.Memcache.Protocol as Memcached
import qualified Network.Memcache as Memcache
import System.Cmd (system)
import System.Locale (defaultTimeLocale, rfc822DateFormat)
import System.Exit (ExitCode (..))
```

It's often useful to have the compactness of the traditional tertiary operator rather than an if then else. The (?) operator can be used like:

```
Bool ? trueExpression $ falseExpression
```

I think I originally saw this on the Haskell wiki.

```
infixl 1 ?  
(?) :: Bool → a → a → a  
(?) = if'
```

```
if' :: Bool → a → a → a  
if' True x _ = x  
if' False _ y = y
```

In case we want don't want our program to crash when taking the head of the empty list, we need to provide a default:

```
safeHead :: a → [a] → a  
safeHead d [] = d  
safeHead _ (x : _) = x
```

Return the current day. I'm not sure that this is useful on its own.

```
currentDay :: IO Day  
currentDay = do  
  now ← getCurrentTime  
  tz ← getCurrentTimeZone  
  let (LocalTime day _) = utcToLocalTime tz now  
  return day
```

Return the current year as a 4-digit number.

```
currentYear :: IO Integer  
currentYear = do  
  day ← currentDay  
  let (year, _, _) = toGregorian day  
  return year
```

Displaying a time is a common enough task.

```
formatSimpleTime :: UTCTime → String  
formatSimpleTime = formatTime defaultTimeLocale "%a %b %d, %Y %R"
```

For files we receive via HTTP, we can't make assumptions about the path separator.

```
basename :: FilePath → FilePath  
basename = reverse ∘ takeWhile (∉ "\\/") ∘ reverse
```

This is like the Unix `tr` utility. It takes a list of search/replacements and then performs them on the list.

```
translate :: (Eq a) ⇒ [(a, a)] → [a] → [a]  
translate sr = map (λs → maybe s id $ lookup s sr)
```

Often it's handy to be able to lift an operation into 2 monads with little verbosity. Parsec may have claimed this operator name before me, but `< $$ >` just makes too much sense as `2 < $ >s`.

```

(< $$ >) :: (Monad m1, Monad m) => (a -> r) -> m (m1 a) -> m (m1 r)
(< $$ >) = liftM o liftM

```

Sending mail is pretty easy. We just deliver it to a local MTA. Even if we have no MTA running locally, there are sendmail emulators that will handle the SMTP forwarding for us so that we don't have to deal with SMTP here.

```

sendMail :: String -> String -> String -> IO (Maybe ())
sendMail to subject body = do
  let body' = unlines ["To: <" ++ to ++ ">",
                      "Subject: " ++ subject,
                      "",
                      body]
      res ← try $ system $
        ("export MAILUSER=vocabulink; "
         "export MAILHOST=vocabulink.com; "
         "export MAILNAME=Vocabulink; "
         "echo -e \"" ++ body' ++ "\" | "
         "sendmail \"" ++ to ++ "\"")
      case res of
        Right ExitSuccess -> return $ Just ()
        _                  -> return Nothing

```

For now, we don't use memcache for anything more than page-level caching. As such, we only need to connect, issue a command, and then disconnect. In the future, if we use it more we could add a memcache server to the App monad.

```

memcacheServer :: IO Memcached.Server
memcacheServer = Memcached.connect "127.0.0.1" 11211

```

```

withMemcache :: (Memcached.Server -> IO a) -> IO a
withMemcache f = do
  bracket (memcacheServer)
          (Memcached.disconnect)
          f

```

```

memcacheSet :: String -> String -> IO (Bool)
memcacheSet key value = withMemcache $ \mc ->
  Memcache.set mc key (encodeString value)

```

```

memcacheDelete :: String -> IO (Bool)
memcacheDelete key = withMemcache $ \mc -> Memcache.delete mc key 0

```

While our dataset and traffic is still small, the easiest thing to do when something is updated is to just flush the entire cache rather than try and figure out what exactly we need to invalidate. It's a brute force approach but it's better than no cache at all.

```

memcacheFlush :: IO ()
memcacheFlush = withMemcache $ \mc -> Memcache.flush mc

```

## 6 CGI

Our program is by nature a CGI application. This is handy and troublesome at the same time. For one, we have to deal with producing some reasonable output for the client of our program, even when given incomplete or incorrect inputs. But, since we're dealing with each of many requests in their own threads, it's not the end of the world if we generate an error or don't catch an exception.

To keep other modules from having to know exactly which CGI method we're using (who knows, we may want to switch to SCGI later), we export some FastCGI functions. This allows other modules to just import `Vocabulink.CGI` and gives us the option of overriding its functions in the future (which we do already with `readInput`). This is a common pattern in other Vocabulink modules.

```
module Vocabulink.CGI (getInput, getRequiredInput, getInputDefault,  
                        readInput, readRequiredInput, readInputDefault,  
                        getInputs, handleErrors', referrerOrVocabulink,  
                        urlify, outputUnauthorized, outputText, outputJSON,  
                        output',  
  {-Network.FastCGI -} getInputFPS, getInputFilename,  
                      MonadCGI, CGIResult, requestURI, requestMethod,  
                      getVar, setHeader, output, redirect, remoteAddr,  
                      outputError, outputMethodNotAllowed,  
                      Cookie (.), getCookie, setCookie, deleteCookie,  
  {-Network.URI -}   uriPath, uriQuery,  
  {-Text.JSON -}    JSON, encode, toJSObject) where  
  
import Vocabulink.DB  
import Vocabulink.Utils  
  
import Control.Exception (Exception (.))  
import Data.ByteString.Lazy.UTF8 (fromString)  
import Data.Char (toLower, isAlphaNum)  
import Network.URI (uriPath, uriQuery)  
import Text.JSON (JSON, encode, toJSObject)
```

We're going to hide some `Network.CGI` functions so that we can override them with versions that automatically handle UTF-8-encoded input.

```
import Network.FastCGI hiding (getInput, readInput, getInputs)  
import Network.CGI.Protocol (CGIResult (.))  
import qualified Network.FastCGI as FCGI
```

It's quite probable that we're going to trigger an unexpected exception somewhere in the program. This is especially likely because we're interfacing with a database via strings. Rather than blow up, we'd like to catch and log the exception before giving the client some sort of indication that something went wrong.

`catchCGI` will handle exceptions in the CGI monad. If no exception is thrown, we'll close the database handle and return the CGI result.

```
handleErrors' :: IConnection conn => conn -> CGI CGIResult -> CGI CGIResult  
handleErrors' c a = catchCGI (do r <- a
```

```

liftIO $ disconnect c
return r)
(outputException' c)

```

Network.CGI provides *outputException* as a basic default error handler. This is a slightly modified version that logs errors.

One case that needs to be tested is when an error message has non-ASCII characters. I'm not sure how *outputInternalServerError* will handle it.

By the time we output the error to the client we no longer need the database handle. This is the perfect place to close it, as it'll be the last thing we do in the CGI monad (and the thread).

```

outputException' :: (MonadCGI m, MonadIO m, IConnection conn) =>
conn -> Exception -> m CGIResult
outputException' c e = do
s <- liftIO $ logException c e
liftIO $ disconnect c
outputInternalServerError [s]

```

Usually we use the *withRequired* functions when an action requires that the client be authenticated. However, sometimes (as with AJAX) we want to output an actual 403 error.

```

outputUnauthorized :: (MonadCGI m, MonadIO m) => m CGIResult
outputUnauthorized = outputError 403 "Unauthorized" []

```

The default *output* method provided by *Network.CGI* does not automatically encode the input.

```

output' :: MonadCGI m => String -> m CGIResult
output' = return o CGIOutput o fromString

```

Also, we do not always output HTML. Sometimes we output JSON or HTML fragments.

```

outputText :: (MonadCGI m, MonadIO m) => String -> m CGIResult
outputText s = setHeader "Content-Type" "text/plain; charset=utf-8" >> output' s

```

To output JSON, we just need an associative list.

```

outputJSON :: (MonadCGI m, MonadIO m, JSON a) => [(String, a)] -> m CGIResult
outputJSON = outputText o encode o toJSONObject

```

In some cases we'll need to redirect the client to where it came from after we perform some action. We use this to make sure that we don't redirect them off of the site.

```

referrerOrVocabulink :: MonadCGI m => m String
referrerOrVocabulink =
fromMaybe "http://www.vocabulink.com/" `liftM` getVar "HTTP_REFERER"

```

We need to handle UTF-8-encoded GET and POST parameters. The following are enhanced versions of Network.CGI's *getInput* and *readInput* along with a few helpers.

```

getInput :: MonadCGI m => String -> m (Maybe String)
getInput = liftM (>>= Just o decodeString) o FCGI.getInput

```

We need to do the same for getInputs. (It's used by *runForm* at the least.)

```

getInputs :: MonadCGI m => m [(String, String)]
getInputs = map decode `liftM` FCGI.getInputs
  where decode (x, y) = (decodeString x, decodeString y)

```

Often we'll want an input from the client but are happy to fall back to a default value.

```

getInputDefault :: MonadCGI m => String -> String -> m String
getInputDefault d p = getInput p >>= return <math>\circ</math> fromMaybe d

```

As a convenience, `getRequiredInput` will throw an error on a missing input. It allows us to write simpler code, but eventually most calls to this should be removed and we should more gracefully handle the error.

```

getRequiredInput :: MonadCGI m => String -> m String
getRequiredInput p =
  getInputDefault (error $ "Parameter '" ++ p ++ "' is required.") p

```

The `Read` versions of the above handle automatically converting the requested input to a required type (as long as that type is `Readable`).

```

readInput :: (Read a, MonadCGI m) => String -> m (Maybe a)
readInput = liftM (>>= maybeRead) <math>\circ</math> getInput

```

```

readInputDefault :: (Read a, MonadCGI m) => a -> String -> m a
readInputDefault d p = readInput p >>= return <math>\circ</math> fromMaybe d

```

```

readRequiredInput :: (Read a, MonadCGI m) => String -> m a
readRequiredInput p =
  readInputDefault (error $ "Parameter '" ++ p ++ "' is required.") p

```

## 6.1 Working with URLs

Certain dynamic parts of the site, such as forum titles, are displayed to the user in a friendly natural form but are also used in URLs. For those cases, it's generally better to use URL-safe representations for both the URL and the natural key in the database. This allows us, for example, to re-title a forum without changing the URL it's located at. Also, since we have the natural key from the URL, we don't need to do an extra database lookup to find the key from a mapping table.

Note that this is not meant to handle arbitrary input from users. For that we can use URL encoding. This is to make common URLs friendly.

Note that this is currently very restrictive until the need arises to permit new characters. It converts spaces to hyphens and only allows alphanumeric characters and hyphens in the resulting string.

```

urlify :: String -> String
urlify = map toLower <math>\circ</math> filter (\e -> isAlphaNum e <math>\vee</math> (e == '-'')) <math>\circ</math> translate [('-', '-')]

```

## 7 The App Monad

When I wrote the first version of Vocabulink, many functions passed around a database connection. I now understand monads a little bit more, and it's easier to store some information within an "App" monad. This reduces our function signatures a little bit.

The App monad is now also used for passing around member information and a few other conveniences.

```
module Vocabulink.App (      App, AppEnv (.), AppT, runApp, logApp, getOption,
                             withMemberNumber, withRequiredMemberNumber,
                             output404, reversibleRedirect,
                             queryTuple', queryValue', queryAttribute',
                             queryTuples', quickInsertNo', runStmt', quickStmt',
                             withTransaction', run',
                             {-Control.Monad.Reader -} asks) where
```

```
import Vocabulink.CGI
import Vocabulink.DB
```

We have to import the authorization token code using GHC's SOURCE directive because of cyclic dependencies.

```
import {-# SOURCE #-} Vocabulink.Member.AuthToken
import Vocabulink.Utils

import Control.Applicative
import Control.Exception (Exception, try)
import Control.Monad (ap)
import Control.Monad.Error (runErrorT)
import Control.Monad.Reader (ReaderT (.), MonadReader, asks)
import Control.Monad.Trans (lift)

import Data.ConfigFile (ConfigParser, get)
import Data.Either.Utils (forceEither)
import Data.List (intercalate)
import Network.CGI.Monad (MonadCGI (.), tryCGI)
import Network.FastCGI (CGI, CGIT, outputNotFound)
import Network.URI (escapeURIStrng, isUnescapedInURI)

data AppEnv = AppEnv { appDB           :: Connection,
                       appCP           :: ConfigParser,
                       appMemberNo     :: Maybe Integer,
                       appMemberName  :: Maybe String,
                       appMemberEmail :: Maybe String }
```

The App monad is a combination of the CGI and Reader monads.

```
newtype AppT m a = AppT (ReaderT AppEnv (CGIT m) a)
deriving (Monad, MonadIO, MonadReader AppEnv)
```

...whose CGI monad uses the IO monad.

```
type App a = AppT IO a
```

We need to make the App monad an Applicative Functor so that it will work with formlets.

```
instance Applicative (AppT IO) where  
  pure = return  
  (< * >) = ap
```

```
instance Functor (AppT IO) where  
  fmap = liftM
```

To make the App monad an instance of MonadCGI, we need to define basic CGI functions. CGI is relatively simple and its functionality can be defined on top of just an environment getter and a function for adding headers. We reuse the existing methods.

```
instance MonadCGI (AppT IO) where  
  cgiAddHeader n v = AppT $ lift $ cgiAddHeader n v  
  cgiGet x = AppT $ lift $ cgiGet x
```

*runApp* does the job of creating the Reader environment and returning the CGIResult from within the App monad to the CGI monad. The environment includes a database handle, a configuration file, and some member information (if the request came from a logged in member).

We can't use the convenience of *getOption* here as we're not in the App monad yet.

```
runApp :: Connection → ConfigParser → App CGIResult → CGI CGIResult  
runApp c cp (AppT a) = do  
  let salt = forceEither $ get cp "DEFAULT" "authtokensalt"  
      token ← verifiedAuthToken salt  
      email ← liftIO $ maybe (return Nothing)  
          (λn → do  
            e ← queryValue c "SELECT email FROM member "  
"WHERE member_no = ?" [toSql n]  
            return $ fromSql < $ > e)  
          (authMemberNo < $ > token)  
      res ← runReaderT a $ AppEnv { appDB           = c,  
                                   appCP           = cp,  
                                   appMemberNo     = authMemberNo 'liftM' token,  
                                   appMemberName   = authUsername 'liftM' token,  
                                   appMemberEmail = email }  
  
  return res
```

At some point it's going to be essential to have all errors and notices logged in 1 location. For now, the profusion of monads and exception handlers makes this difficult. *logApp* will write a message to the database. It takes a type name which are enumerated in the database.

```
logApp :: String → String → App (String)  
logApp type' message = do  
  c ← asks appDB  
  liftIO $ logMsg c type' message
```

## 7.1 Convenience Functions

Here are some functions that abstract away even having to ask for values from the Reader environment in the App monad.

### 7.1.1 Identity

*withMemberNumber* accepts a default action (for if the client isn't logged in) and a function to carry out with the member's number otherwise.

```
withMemberNumber :: a → (Integer → App a) → App a
withMemberNumber d f = asks appMemberNo ≧ maybe (return d) f
```

*withRequiredMemberNumber* is like *withMemberNumber*, but it also checks to see if the member has confirmed their email address and provides a “logged out default” of redirecting the client to the login page.

Use this any time a member number is generally required. If you only use *withMemberNumber* you will allow actions from unconfirmed members.

```
withRequiredMemberNumber :: (Integer → App CGIResult) → App CGIResult
withRequiredMemberNumber f = do
  memberNo ← asks appMemberNo
  email ← asks appMemberEmail
  case (memberNo, email) of
    (Just mn, Just _) → f mn
    (Just _, Nothing) → redirect ≪ reversibleRedirect "/member/confirmation"
    _ → redirect ≪ reversibleRedirect "/member/login"
```

When we direct a user to some page, we might want to make sure that they can find their way back to where they were. To do so, we get the current URI and append it to the target page in the query string. The receiving page might know what to do with it.

```
reversibleRedirect :: String → App String
reversibleRedirect path = do
  request ← fromMaybe "/" 'liftM' getVar "REQUEST_URI"
  return $ path ++ "?redirect=" ++ escapeURIStr isUnescapedInURI request
```

We want to log 404 errors in the database, as they may indicate a problem or opportunity with the site. This takes a list of Strings that are stored in the log. It outputs to the user the requested URI.

```
output404 :: [String] → App CGIResult
output404 s = do logApp "404" (show s)
                 outputNotFound $ intercalate "/" s
```

### 7.1.2 Database

When we're dealing with the database, there's always a chance we're going to have some sort of error (there's a seemingly infinite number of possible sources). We don't want the entire page to blow up if there are errors. Also, we don't really care what the cause of the error is at the time of execution. SQL errors are not something we can generally recover from. We just

need to log the error, return some sort of error indicator to the calling function (in this case, `Nothing`), and get on with it.

In many cases, the calling function will still need to do data validation anyway (make sure that a list of the expected size is returned, etc), so the extra `Maybe` wrapper shouldn't be much extra trouble. In fact, in some cases it's much easier than manually wrapping the query with `catchSql`.

Don't worry about understanding these definitions until you've read through the DB module.

Maybe there's some way to cut down on this code with template Haskell or somesuch, but it works for now.

```
queryTuple' :: String → [SqlValue] → App (Maybe [SqlValue])
queryTuple' sql vs = do
  c ← asks appDB
  liftIO $ (queryTuple c sql vs ≧ return ∘ Just) 'catchSqlD' Nothing
```

```
queryTuples' :: String → [SqlValue] → App (Maybe [[SqlValue]])
queryTuples' sql vs = do
  c ← asks appDB
  liftIO $ (quickQuery' c sql vs ≧ return ∘ Just) 'catchSqlD' Nothing
```

```
queryValue' :: String → [SqlValue] → App (Maybe SqlValue)
queryValue' sql vs = do
  c ← asks appDB
  liftIO $ (queryValue c sql vs) 'catchSqlD' Nothing
```

```
queryAttribute' :: String → [SqlValue] → App (Maybe [SqlValue])
queryAttribute' sql vs = do
  c ← asks appDB
  liftIO $ (queryAttribute c sql vs ≧ return ∘ Just) 'catchSqlD' Nothing
```

```
quickInsertNo' :: String → [SqlValue] → String → App (Maybe Integer)
quickInsertNo' sql vs seqname = do
  c ← asks appDB
  liftIO $ quickInsertNo c sql vs seqname 'catchSqlD' Nothing
```

```
runStmt' :: String → [SqlValue] → App (Maybe Integer)
runStmt' sql vs = do
  c ← asks appDB
  liftIO $ (run c sql vs ≧ return ∘ Just) 'catchSqlD' Nothing
```

It may seem strange to return `Maybe ()`, but we want to know if the database change succeeded.

```
quickStmt' :: String → [SqlValue] → App (Maybe ())
quickStmt' sql vs = do
  c ← asks appDB
  liftIO $ (quickStmt c sql vs ≧ return ∘ Just) 'catchSqlD' Nothing
```

Working with transactions outside of the App monad can be done, but we might as well make a version that fits with the rest of the style of the program (logs the exception and returns Nothing).

```

withTransaction' :: App a → App (Maybe a)
withTransaction' actions = do
  c ← asks appDB
  r ← tryApp actions
  case r of
    Right x → do liftIO $ commit c
                  return $ Just x
    Left e  → do logApp "exception" $ show e
                  liftIO $ try (rollback c) -- Discard any exception here
                  return Nothing

run' :: String → [SqlValue] → App (Integer)
run' sql vs = do
  c ← asks appDB
  liftIO $ run c sql vs

```

### 7.1.3 Exceptions

`tryApp` is like `tryCGI`. It allows us to catch exceptions within the App monad. To do so, we unwrap the Reader monad and use `tryCGI` (which unwraps another Reader and Writer).

```

tryApp :: App a → App (Either Exception a)
tryApp (AppT c) = AppT (ReaderT (λr → tryCGI (runReaderT c r)))

```

### 7.1.4 Configuration

Return a configuration option or log an error.

This always pulls from the `DEFAULT` section. It also only supports strings.

```

getOption :: String → App (Maybe String)
getOption option = do
  cp ← asks appCP
  opt ← runErrorT $ get cp "DEFAULT" option
  case opt of
    Left e  → logApp "config" (show e) >> return Nothing
    Right o → return $ Just o

```

## 8 Database

Vocabulink makes heavy use of PostgreSQL. We use `Database.HDBC` for interfacing with it.

This module is possibly the most dangerous and error-prone we have. Because we interface with the database via strings, we can't do type checking. We're also dealing with ever-changing state.

```

module Vocabulink.DB (
    queryTuple, queryValue, queryAttribute,
    quickStmt, insertNo, quickInsertNo,
    catchSqlD, catchSqlE, logMsg, logException,
    connect,
    {-Database.HDBC -} SqlValue (.), toSql, fromSql, iToSql,
    withTransaction, throwDyn,
    quickQuery, quickQuery',
    IConnection (.), execute, catchSql,
    {-Database.HDBC.PostgreSQL -} Connection) where

```

We need to keep this module independent of most other modules as most modules need to utilize the database in some way.

```

import Vocabulink.Utils

```

```

import Control.Exception (Exception (.), IOException, bracket, throwDyn)
import Database.HDBC
import Database.HDBC.PostgreSQL (connectPostgreSQL, Connection)
import System.IO.Error (isUserError, ioeGetErrorString)

```

Here's how we establish a connection to the database. I'd like to have the database password stored in the configuration file, but it would make the code far more complex.

```

connect :: IO Connection
connect = connectPostgreSQL "host=localhost "
    "dbname=vocabulink "
    "user=vocabulink "
    "password=phae9Xom"

```

## 8.1 Query Helpers

HDBC provides a pretty basic interface. If we relied on it, we'd be doing a lot of housekeeping and repetitive work throughout the code. Here are some higher-level interfaces to the database.

Sometimes we want just the first tuple of a query result. If the query returns multiple tuples, all but the first will be silently discarded.

```

queryTuple :: IConnection conn => conn -> String -> [SqlValue] -> IO [SqlValue]
queryTuple c sql vs = safeHead [] 'liftM' quickQuery' c sql vs

```

Sometimes we just want to retrieve a single attribute from a single tuple. This will return either Just the value you were expecting or Nothing.

```

queryValue :: IConnection conn => conn -> String -> [SqlValue] -> IO (Maybe SqlValue)
queryValue c sql vs = do
    t ← queryTuple c sql vs
    return $ case t of
        [SqlNull] -> Nothing
        [x]        -> Just x
        -          -> Nothing

```

And finally, sometimes we just want to retrieve a single attribute for multiple tuples. This assumes that the attribute you want is the first one SELECTed.

```
queryAttribute :: IConnection conn => conn -> String -> [SqlValue] -> IO [SqlValue]
queryAttribute c sql vs = map head 'liftM' quickQuery' c sql vs
```

It's often tedious to work with transactions if you're just issuing a single statement.

```
quickStmt :: IConnection conn => conn -> String -> [SqlValue] -> IO ()
quickStmt c' sql vs = withTransaction c' $ \c -> run c sql vs >> return ()
```

A common task is to insert a tuple and get its sequence number. This should only be used for inserting into tables with a sequence-based primary key (e.g. the SERIAL type).

```
insertNo :: IConnection conn =>
  conn -> String -> [SqlValue] -> String -> IO (Maybe Integer)
insertNo c sql vs seqName = do
  run c sql vs
  seqNo <- queryValue c "SELECT currval(?)" [toSql seqName]
  return $ fmap fromSql seqNo
```

This is the same as *insertNo*, but with its own transaction.

```
quickInsertNo :: IConnection conn =>
  conn -> String -> [SqlValue] -> String -> IO (Maybe Integer)
quickInsertNo c' sql vs seqName =
  withTransaction c' $ \c -> insertNo c sql vs seqName
```

## 8.2 Error Handling

Most of the time, if we have a SQL error, we're not prepared for it. We want to log it and fail with some message to the user. We hope that we don't ever end up in this code, because it blows up the entire HTTP request. It's here only for unrecoverable situations and its only purpose is to log the error and hide the gory details from the client.

For now, we should use the following catches after every database query or statement we execute. It's tedious, but that's what we have to deal with if we want access to a database. There may be a better way of handling errors, but I haven't found it yet.

The most common cause for triggering a database exception is through malformed SQL, so we probably find most of these during early testing.

```
catchSqlE :: IO a -> String -> IO a
catchSqlE sql msg = catchSqlD sql (error msg)
```

Instead of erroring out, it might make more sense to return a default value. When we don't want the entire request crashing, this is a better alternative to *catchSqlE*.

This is a little bit wasteful, but if we do catch an exception we establish a new database connection. We do this so that we don't have to pass around a database connection. But also, we may have been in a transaction or otherwise ruined our main connection and need a new one for logging the error message. Establishing an extra connection shouldn't be too much extra trouble: we've already encountered an error condition, how much worse can it get?

```

catchSqlD :: IO a → a → IO a
catchSqlD sql d = sql `catchSql` (λe → bracket (connect)
  (disconnect)
  (λc → do logSqlError c e
    return d))

```

It's useful to have all errors logged in 1 location: the database.

`logMsg` takes a log type name ("SQL exception", "404", etc.) and a descriptive message. The type and message is then logged to the database along with a timestamp. If the message type is not found, it defaults to "unknown".

```

logMsg :: IConnection conn ⇒ conn → String → String → IO (String)
logMsg c t s = do
  quickStmt c "INSERT INTO log (type, message) "
  "VALUES ((SELECT name FROM log_type WHERE name = ?), ?)"
  [toSql t, toSql s]
  `catchSqlD` ()
  return s

```

Exceptions come in different shapes and sizes, and we'd like to have log information about them when we encounter them. Or, we might want to ignore certain exceptions.

```

logException :: IConnection conn ⇒ conn → Exception → IO (String)
logException c e =
  case sqlExceptions e of
    Nothing → case e of
      (ErrorCall msg) → logMsg c "exception" msg
      (IOException ie) → logMsg c "IO exception" $
        readableIOException ie
      e' → logMsg c "exception" (show e')
    Just se → do logSqlError c se
      return "Database Error"

```

When we encounter an IO exception, we'd like to pull the information out of it so that we can make sense of it when going through the logs.

```

readableIOException :: IOException → String
readableIOException ioe = isUserError ioe ? ioeGetErrorString ioe $ show ioe

```

`logSqlError` is used by `logException`, `catchSqlD`, and `catchSqlE`. It's special because it's potentially the most common type of exception we'll encounter, and the one we most want to hear about.

```

logSqlError :: IConnection conn ⇒ conn → SqlError → IO (String)
logSqlError c se = logMsg c "SQL error" (init (seErrorMsg se))

```

## 9 Html

Much of Vocabulink consists of simple, program-generated HTML. Rather than use templates or HTML in strings, we use an HTML combinator library (Text.XHtml.Strict). This makes it

almost certain that our HTML will be well formed (although we have no guarantee that it will be valid). But more importantly, it allows us to use abstraction to get higher-level HTML-based functions. An example of this is *linkList*.

```

module Vocabulink.Html (Dependency (.), stdPage, simplePage,
    linkList, breadcrumbs, options, tableRows, accesskey,
    helpButton, markdownToHtml,
    AppForm, runForm, runForm', formLabel, formLabel',
    checkbox', tabularInput, tabularSubmit,
    pager, currentPage,
    {-Text.XHtml.Strict -} Html, noHtml, primHtml, stringToHtml, concatHtml,
    (<<), (+ + +), (!), showHtmlFragment,
    identifier, theclass, thediv, thespan, style,
    paragraph, pre, h1, h2, h3, br, anchor, href, script,
    image, unordList, form, action, method, enctype,
    hidden, label, textfield, password, button, submit,
    fieldset, legend, afile, textarea, select, widget,
    thestyle, src, width, height, value, name,
    cols, rows, colspan, caption,
    table, thead, tbody, tfoot, th, tr, td,
    {-Text.Formlets -} runFormState, nothingIfNull,
    check, ensure, ensures, checkM, ensureM,
    plug,
    {-Text.XHtml.Strict.Formlets -} XHtmlForm) where

```

```

import Vocabulink.App
import Vocabulink.CGI
import Vocabulink.Review.Html
import Vocabulink.Utils

```

```

import Control.Arrow (second)
import Data.List (intersperse, find)
import Text.Regex (mkRegex, subRegex)
import Text.Regex.Posix ((= ~))
import Text.Formlets (runFormState, plug, nothingIfNull,
    check, ensure, ensures, checkM, ensureM)
import Text.Pandoc (readMarkdown, writeHtml, defaultParserState,
    defaultWriterOptions)
import Text.Formlets as F
import Text.XHtml.Strict
import Text.XHtml.Strict.Formlets (XHtmlForm)

```

Most pages depend on some external CSS and/or JavaScript files.

```

data Dependency = CSS String | JS String

```

*stdPage* takes a title, a list of dependencies, and list of HTML objects to place into the body of the page. It automatically adds a standard header and footer. It also includes `page.css` and conditionally includes an Internet Explorer-specific stylesheet for the few cases when there's no

other way to work around a defect in Internet Explorer that would otherwise seriously impact usability.

If any JavaScript files are required, *stdPage* will automatically add a `<noscript>` warning to the top of the page.

*stdPage* also stores output to memcache for working with nginx's memcache module. It only stores a document in memcache when requested by a non-member (we don't want private details or customized views being cached). We also only want to cache GET requests. Also, so that we don't fill up the cache with previews and such, we don't cache anything with a query string.

On the nginx side, we check to see if the incoming request is a GET, has no query string, and is does not have an auth cookie. Only if all 3 conditions are met do we serve the file from the cache.

This is not a good long-term caching strategy. But it's nice and simple and gives us good control for the time being.

```

stdPage :: String → [Dependency] → [Html] → [Html] → App CGIResult
stdPage title' deps head' body' = do
  headerB ← headerBar
  footerB ← footerBar
  setHeader "Content-Type" "text/html; charset=utf-8"
  let xhtml = renderHtml $ header <<
      [thetitle << title',
        concatHtml (map includeDep ([CSS "page"] ++ deps)),
        primHtml "<!-- [if IE]>"
      ]
      "<link rel=\"stylesheet\" type=\"text/css\" \"
      \"href=\"http://s.vocabulink.com/css/ie.css\" />"
      "<![endif]->",
      concatHtml head'] ++ +
      body << [headerB,
              jsNotice,
              thediv ! [identifier "body"] << concatHtml body',
              footerB]
  memberNo ← asks appMemberNo
  uri      ← requestURI
  method' ← requestMethod
  liftIO $ case (memberNo, method', uriQuery uri) of
    (Nothing, "GET", "") → do
      memcacheSet ("vocabulink.com:" ++ uriPath uri) xhtml
      return ()
    _                    → return ()
  output' xhtml
  where jsNotice = case find (λe → case e of
    JS _ → True
    _    → False) deps of
    Nothing → noHtml
    Just _  → noscript << paragraph <<
      "This page requires JavaScript for some functionality."

```

Often we just need a simple page where the title and header are the same.

```

simplePage :: String → [Dependency] → [Html] → App CGIResult
simplePage t deps h = stdPage t deps [] $ [h1 << t] ++ h

```

Each dependency is expressed as the path from the root of the static files subdomain (for now, `s.vocabulink.com`) to the file. Do not include the file suffix (`.css` or `.js`); it will be appended automatically. These are meant for inclusion in the `<head>` of the page.

```

includeDep :: Dependency → Html
includeDep (CSS css) =
  thelink ! [href ("http://s.vocabulink.com/css/" + css + ".css"),
    rel "stylesheet", thetype "text/css"] << noHtml
includeDep (JS js) =
  script ! [src ("http://s.vocabulink.com/js/" + js + ".js"),
    thetype "text/javascript"] << noHtml

```

The standard header bar shows the Vocabulink logo (currently just some text), a list of hyperlinks, a search box, and either a login/sign up button or a logout button. If the page is being served to a logged-in member it also includes a notice about the number of links that the member has waiting for review.

```

headerBar :: App Html
headerBar = do
  username ← asks appMemberName
  review ← reviewBox
  return $ thediv ! [identifier "header-bar"] << [
    [anchor ! [theClass "logo", href "/", accesskey "1"] << [
      stringToHtml "Vocabulink", br,
      thespan ! [theClass "tagline"] << "learn languages through fiction"],
    topLinks,
    maybe loginBox logoutBox username,
    searchBox,
    review,
    thediv ! [theClass "clear"] << noHtml]

```

Here are the hyperlinks we want in the header of every page.

```

topLinks :: Html
topLinks = linkList
  [anchor ! [href "/forums"] << "Forums",
  anchor ! [href "/articles"] << "Articles",
  anchor ! [href "/links"] << "Latest Links",
  anchor ! [href "/help"] << "Help"]

```

The footer bar is more simple. It just includes some hyperlinks to static content.

```

footerBar :: App Html
footerBar = do
  copy ← copyrightNotice
  return $ thediv ! [identifier "footer-bar"] << [
    linkList
    [anchor ! [href "/help"] << "help",

```

```

    anchor![href "/privacy"] << "privacy policy",
    anchor![href "/terms-of-use"] << "terms of use",
    anchor![href "/source"] << "source",
    copy,
    googleAnalyticsTag]

```

We want a copyright notice at the bottom of every page. Since this is a copyright notice for dynamic content, we want it to be up-to-date with the generation time (now).

```

copyrightNotice :: App Html
copyrightNotice = do
  year ← liftIO currentYear
  return $ paragraph![theclass "copyright"] <<
    [stringToHtml "Copyright 2008-",
     stringToHtml ((show year) ++ " "),
     anchor![href "http://jekor.com/"] << "Chris Forno"]

```

We use Google Analytics for tracking site usage. It requires the JavaScript tag to be placed on every page.

```

googleAnalyticsTag :: Html
googleAnalyticsTag = primHtml $ unlines [
  "<script type=\"text/javascript\">",
  "var gaJsHost = ((\"https:\" == document.location.protocol) ?"
  " \"https://ssl.\" : \"http://www.\");",
  "document.write(unescape(\"%3Cscript src=\'\" + gaJsHost "
  "+ \"google-analytics.com/ga.js\" "
  "type='text/javascript'%3E%3C/script%3E\"));",
  "</script>",
  "<script type=\"text/javascript\">",
  "try {",
  "pageTracker = _gat._getTracker(\"UA-73938-2\");",
  "pageTracker._trackPageview();",
  "} catch(err) {}</script>"]

```

The following are just login and signup buttons.

```

loginBox :: Html
loginBox = thespan![theclass "auth-box login"] << [
  anchor![href "/member/login"] << "Login", stringToHtml " | ",
  anchor![href "/member/signup"] << "Sign Up"]

```

For logged-in members, we provide a logout button (with an indicator of your username to show that you're logged in).

```

logoutBox :: String → Html
logoutBox username = form![theclass "auth-box logout", action "/member/logout",
                           method "POST"] <<
  [stringToHtml username, submit "" "Log Out"]

```

Students with a goal in mind will want to search for words they're studying rather than browse randomly. We display a search box for them at the top of the page. This also is currently the

only way to create new links (aside from entering in the URL manually), but that might change in the future.

```
searchBox :: Html
searchBox = form ![theClass "search-box", action "/links", method "GET"] <<
  [textField "contains" ![accessKey "s"], stringToHtml " ",
    submit "" "Search Links"]
```

## 9.1 Higher-Level Combinators

It's common to use an unordered list to present a series of hyperlinks. For example, both the standard header and footer use this.

```
linkList :: (HTML a) => [a] -> Html
linkList items = ulist ![theClass "hyperlinks"] << map (li<<) items
```

Breadcrumbs are a common navigation element. This only handles wrapping the provided elements in an appropriate ordered list and adding decorations. Adding the anchors is up to you.

```
breadcrumbs :: [Html] -> Html
breadcrumbs items = ulist ![theClass "breadcrumbs"] << map (li<<) items'
  where items' = intersperse (stringToHtml " » ") items
```

Sometimes you just want a select list where the displayed options match their values.

```
options :: [String] -> [Html]
options choices = [option ![value choice] << choice | choice ← choices]
```

This automatically adds “odd” and “even” CSS classes to each table row.

```
tableRows :: [Html] -> [Html]
tableRows = map decorate ∘ zip [1..]
  where decorate (a, b) = tr ![theClass (odd (a :: Integer) ? "odd" $ "even")] << b
```

Curiously, the accesskey attribute is missing from Text.XHtml.

```
accesskey :: String -> HtmlAttr
accesskey = strAttr "accesskey"
```

It's nice to have little help buttons and such where necessary. Making them easier to create means that we're more likely to do so, which leads to a more helpful user interface.

Currently this uses an icon from the FamFamFam “Mini” set (<http://www.famfamfam.com/lab/icons/mini/>).

```
helpButton :: String -> Maybe String -> Html
helpButton url label' = anchor ![href url, theClass "button"] << [
  image ![src "http://s.vocabulink.com/icon_info.gif"],
  maybe noHtml (λx -> stringToHtml $ " " ++ x) label']
```

## 9.2 Other Markup

A modified version of Markdown (Pandoc Markdown) is used in comments and link bodies.

```
markdownToHtml :: String → Html
markdownToHtml = (writeHtml defaultWriterOptions) ∘
  readMarkdown defaultParserState
```

## 9.3 Form Builders

For complex forms, we use tables. Tables allow for proper alignment that makes the form much easier to read. This type of form tends to have a number of common elements that we can abstract out.

One thing that we're don't currently do is hook the label to the control using the "for" attribute.

```
tabularInput :: String → Html → Html
tabularInput l i = tr << [th << (label << (l ++ ":")),
  td << i]
```

We want any submit button centered on a row of its own.

```
tabularSubmit :: String → Html
tabularSubmit l = tr << td ![colspan 2] << submit "" l
```

## 9.4 Formlet Helpers

Formlets are a great tool for abstracting and building complex forms. But the library is still a bit rough around the edges. These helpers are by no means elegant, but they help get the job done.

All of the formlets we build have this type:

```
type AppForm a = XHtmlForm (AppT IO) a
```

We offer want to "wrap" a label around a form component. Note that this doesn't currently set a for attribute either.

```
formLabel :: Monad m ⇒ String → XHtmlForm m a → XHtmlForm m a
formLabel text = plug (λxhtml → label << (text ++ ": ") + + + xhtml)
```

Here's an alterate version of the above which also adds a paragraph.

```
formLabel' :: Monad m ⇒ String → XHtmlForm m a → XHtmlForm m a
formLabel' text = plug (λxhtml → paragraph << (label << (text ++ ": ") + + + xhtml))
```

Curiously, the formlets library is missing a checkbox implementation. Thanks to Chris Done (<http://chrisdone.com/blog/html/2008-12-14-haskell-formlets-composable-web-form-construction-and-validation.html>) for this one.

```
checkbox' :: Monad m ⇒ String → XHtmlForm m (Maybe String)
checkbox' l = optionalInput box where
  box name' = input ![thetype "checkbox", name name'] + + + l
```

We use `runForm` for most of the heavy lifting. It takes a form and a submit button label, runs the form, and returns either the form to display (with errors, if any) or the result of the form.

“Running” the form involves taking the form inputs from the “environment” (the CGI input variables) and “passing” them to the form. The form then attempts to validate against the environment. If it fails, it returns a form (as `Html`) to display to the client, but if it succeeds it returns a value of the type of the form.

`s` is either a label or some custom `Html` for the submit button (or `noHtml` if you don’t want a submit button).

```
runForm :: XHtmlForm (AppT IO) a → Either String Html → App (Either Html a)
runForm frm s = do
  (status, xhtml) ← runForm' frm
  case status of
    Failure failures → do
      uri ← requestURI
      meth ← requestMethod
      let submit' = case s of
          Left s' → submit "" s'
          Right h → h
          return $ Left $ form ![action (uriPath uri), method "POST"] <<
            [(meth ≡ "GET" ? noHtml $ unordList failures),
             xhtml, submit']
      Success result → return $ Right result
```

This is a slimmer wrapper around `runFormState` for when you want to get access to the errors before they’re packed into the returned `Html`. This is also handy when implementing “preview” functionality for forms.

```
runForm' :: XHtmlForm (AppT IO) a → App (Failing a, Html)
runForm' frm = do
  env ← map (second Left) < $ > getInputs
  let (res, markup, _) = runFormState env "" frm
      status ← res
      xhtml ← markup
      return (status, xhtml)
```

## 9.5 Paging

We’d like to have a consistent way of “paging” lists that don’t fit on a single page. This can be used to page search results, a set of links, articles, etc.

This reads the page query parameters and returns them along with the current offset (as a convenience). In the absence of certain parameters, we fall back to reasonable defaults like 10 items per page. We also don’t want to chew up resources retrieving too many items, so we cap the max that a client can request at 100.

Limiting the paging elements to `Int` bounds is necessary for the functions that use the pager (they often need to *take* some number of tuples from a list, for instance) and does not limit the design of our HTTP interface much, if at all. I cannot think of an instance where we’d need to go past the 65,000th page of anything.

```
currentPage :: App (Int, Int, Int)
currentPage = do
```

```

pg ← readInputDefault 1 "pg"
n' ← readInputDefault 10 "n"
let n'' = n' > 100 ? 100 $ n'
    n    = n'' < 1 ? 1 $ n''
    offset = (pg - 1) * n
return (pg, n, offset)

```

This will handle the query string in the hyperlinks it generates while it replaces the `n` (number of items per page) and `page` (the page we're on) parameters. We give it the page we're currently on, the number of items per page, and the total number of items available, and it does the rest.

This doesn't actually display clickable numeric hyperlinks such as you'd see on a Google search results page. It only provides the client with "previous" and "next". We do this because determining the number of pages in a result can be expensive.

```

pager :: Int → Int → Int → App Html
pager pg n total = do
  q' ← getVar "QUERY_STRING"
  uri ← requestURI
  let path = uriPath uri
      q    = maybe "" decodeString q'
      prev = pageQueryString n (pg - 1) q
      next = pageQueryString n (pg + 1) q
  return $ paragraph ! [theClass "pager"] << thespan ! [theClass "controls"] <<
    [ (pg > 1 ? anchor ! [href (path ++ prev), theClass "prev"] $
      thespan ! [theClass "prev"]) << "Previous", stringToHtml " ",
      ((pg * n < total) ? anchor ! [href (path ++ next), theClass "next"] $
        thespan ! [theClass "next"]) << "Next" ]

```

Creating the query string involves keeping the existing query string intact as much as possible. We even want the position of the parameters to stay the same if they're already there.

```

pageQueryString :: Int → Int → String → String
pageQueryString n pg q =
  let q1 = q = ~ nRegex ? subRegex (mkRegex nRegex) q ("n=" ++ show n) $
        q ++ "&n=" ++ show n
      q2 = q1 = ~ pgRegex ? subRegex (mkRegex pgRegex) q1 ("pg=" ++ show pg) $
        q1 ++ "&pg=" ++ show pg
  in "?" ++ q2
  where nRegex = "n=[^&]+"
        pgRegex = "pg=[^&]+"

```

## 10 Authentication Tokens

Much of what you can do on Vocabulink requires us to know and trust who you say you are.

```

module Vocabulink.Member.AuthToken (AuthToken (.), verifiedAuthToken,
                                     setAuthCookie, deleteAuthCookie) where

```

```

import Vocabulink.App
import Vocabulink.CGI
import Vocabulink.Utils

```

```

import Data.ByteString.Char8 (pack)
import Data.Digest.OpenSSL.HMAC (hmac, sha1)
import Data.Time.Calendar (addDays, diffDays, showGregorian)
import Data.Time.Format (parseTime)
import System.Locale (defaultTimeLocale, iso8601DateFormat)
import System.Time (TimeDiff (..), getClockTime, addToClockTime, toCalendarTime)
import Network.URI (escapeURIString, unEscapeString, isUnescapedInURI)
import Text.ParserCombinators.Parsec (Parser, parse, manyTill, many1,
                                       anyChar, char, string)

```

## 10.1 Creating the Auth Token

Each time a member logs in, we send an authentication cookie to their browser. The cookie is a digest of some state information. We then use the cookie for authenticating their identity on subsequent requests.

We don't want to associate personally-identifying information like IP addresses to member names in our database. However, we do need to know which IP address the member logged in from so that we can protect against request spoofing. The solution is to store all of the session state stays in the cookie. This also means we don't have to deal with storing session state on our end.

```

data AuthToken = AuthToken {
  authExpiry    :: Day,
  authMemberNo :: Integer,
  authUsername  :: String,
  authIPAddress :: String,
  authDigest    :: String
}

```

We give the token 30 days to expire. We don't want it expiring too soon because it becomes bothersome to keep logging in. However, we don't want members to stay logged in forever, and using a non-expiring cookie would require us to check the database for each authentication to see if we need to de-authenticate the client.

```

cookieShelfLife :: Integer
cookieShelfLife = 30

```

Here is the format of the actual cookie we send to the client.

```

instance Show AuthToken where
  show a = "exp=" ++ showGregorian (authExpiry a) ++
    "&no=" ++ show (authMemberNo a) ++
    "&name=" ++ escapeURIString isUnescapedInURI
              (encodeString $ authUsername a) ++
    "&ip=" ++ authIPAddress a ++
    "&mac=" ++ authDigest a

```

This creates an AuthToken with the default expiration time, automatically calculating the digest.

```

authToken :: Integer → String → String → String → IO (AuthToken)
authToken memberNo username ip salt = do
  now ← currentDay
  let expires = addDays cookieShelfLife now
      digest ← tokenDigest (AuthToken { authExpiry    = expires,
                                         authMemberNo = memberNo,
                                         authUsername  = username,
                                         authIPAddress = ip,
                                         authDigest    = "" }) salt
  return AuthToken { authExpiry    = expires,
                    authMemberNo = memberNo,
                    authUsername  = username,
                    authIPAddress = ip,
                    authDigest    = digest }

```

This generates the HMAC digest of the auth token using SHA1.

Eventually, we need to rotate the key used to generate the HMAC, while still storing old keys long enough to use them for any valid login session. Without this, authentication is less secure.

```

tokenDigest :: AuthToken → String → IO (String)
tokenDigest a salt = hmac sha1 (pack salt) (pack token)
  where token = showGregorian (authExpiry a) ++
               show (authMemberNo a) ++
               encodeString (authUsername a) ++
               authIPAddress a

```

Setting the cookie is rather simple by this point. We just create the auth token and send it to the client.

```

setAuthCookie :: Integer → String → String → App ()
setAuthCookie memberNo username ip = do
  salt ← fromJust <$ > getOption "authtokensalt"
  authTok ← liftIO $ authToken memberNo username ip salt
  now ← liftIO getClockTime
  expires ← liftIO $ toCalendarTime
    (addToClockTime (TimeDiff { tdYear    = 0,
                                tdMonth   = 0,
                                tdDay     = fromIntegral cookieShelfLife,
                                tdHour    = 0,
                                tdMin     = 0,
                                tdSec     = 0,
                                tdPicosec = 0 }) now)
  setCookie Cookie { cookieName  = "auth",
                    cookieValue = show authTok,
                    cookieExpires = Just expires,
                    cookieDomain = Just "vocalink.com",
                    cookiePath  = Just "/",
                    cookieSecure = False }

deleteAuthCookie :: MonadCGI m ⇒ m ()
deleteAuthCookie =

```

```

deleteCookie Cookie { cookieName = "auth",
                      cookieDomain = Just "vocalink.com",
                      cookiePath = Just "/",
                      -- The following are only here to get rid of GHC warnings.
                      cookieValue = "",
                      cookieExpires = Nothing,
                      cookieSecure = False }

```

## 10.2 Reading the Auth Token

This retrieves the auth token from the HTTP request, verifies it, and if valid, returns it. To verify an auth token, we verify the token digest, check that the cookie hasn't expired, and check the requestor's IP address against the one in the token.

```

verifiedAuthToken :: (MonadCGI m, MonadIO m) => String -> m (Maybe AuthToken)
verifiedAuthToken salt = do
  cookie <- getCookie "auth"
  ip <- remoteAddr
  case parseAuthToken <=< cookie of
    Nothing -> return Nothing
    Just a -> do
      now <- liftIO currentDay
      digest <- liftIO $ tokenDigest a salt
      if digest == authDigest a ^& diffDays (authExpiry a) now > 0 ^&
         ip == (authIPAddress a)
      then return $ Just a
      else return Nothing

```

This is a Parsec parser for auth tokens (as stored in cookies).

```

authTokenParser :: Parser (Maybe AuthToken)
authTokenParser = do
  string "exp="; day' <- manyTill anyChar $ char '&'
  string "no="; memberNo <- manyTill anyChar $ char '&'
  string "name="; username <- manyTill anyChar $ char '&'
  string "ip="; ip <- manyTill anyChar $ char '&'
  string "mac="; digest <- many1 anyChar
  let day = parseTime defaultTimeLocale (iso8601DateFormat Nothing) day'
  return $ day >=> \d -> Just AuthToken { authExpiry = d,
                                         authMemberNo = read memberNo,
                                         authUsername = decodeString $
                                                         unEscapeString username,
                                         authIPAddress = ip,
                                         authDigest = digest }

```

It'd be nice to log an error if parsing fails, but we don't have a database handle.

```

parseAuthToken :: String -> Maybe AuthToken
parseAuthToken s = case parse authTokenParser "" s of
  Left _ -> Nothing
  Right x -> x

```

## 11 Members

Most functionality on Vocabulink—such as review scheduling—is for registered members only.

```
module Vocabulink.Member (login, logout, registerMember,  
                          getMemberNumber, getMemberName,  
                          confirmEmail, confirmEmailPage,  
                          memberSupport) where
```

```
import Vocabulink.App  
import Vocabulink.CGI  
import Vocabulink.DB  
import Vocabulink.Html  
import Vocabulink.Member.AuthToken  
import Vocabulink.Utils
```

```
import qualified Text.XHtml.Strict.Formlets as F
```

### 11.1 Authentication

To authenticate a member, we need their username and password. This is our first example of a formlet. The formlet nicely encapsulates validating the member's password against the database as part of normal formlet validation.

```
loginForm :: String → AppForm (String, String)  
loginForm ref = plug (λxhtml → hidden "redirect" ref + + +  
                    table <<  
                      (xhtml + + + tfoot << tabularSubmit "Login"))  
  ((,) < $ > username < * > passwd "Password") `checkM`  
  ensureM passMatch err  
where passMatch (u, p) = do  
  valid ← queryValue' "SELECT password_hash = crypt(?, password_hash) "  
"FROM member WHERE username = ?"  
  [toSql p, toSql u]  
  return $ case valid of  
    Nothing → False  
    Just v → fromSql v  
  err = "Username and password do not match (or don't exist)."
```

If a member authenticates correctly, we redirect them to either the frontpage or whatever redirect parameter we've been given. This allows *withRequiredMemberNumber* to function properly: getting the client to login and then continuing where it left off.

```
login :: App CGIResult  
login = do  
  ref ← referrerOrVocabulink  
  redir ← getInputDefault ref "redirect"  
  res ← runForm (loginForm redir) $ Right noHtml  
case res of
```

```

Left xhtml → simplePage "Login" []
  [paragraph! [thestyle "text-align: center"] <<
    [stringToHtml "Not a member? ",
      anchor! [href "/member/signup"] << "Sign Up for free!"],
    xhtml]
Right (user, _) → do
  ip ← remoteAddr
  memberNo ← getMemberNumber user
  case memberNo of
    Nothing → redirect redir
    Just n → do
      setAuthCookie n user ip
      redirect redir

```

At the time of authentication, we have to fetch the member's number from the database before it can be packed into their auth token. There may be a way to put this step into password verification so that we don't need 2 queries.

```

getMemberNumber :: String → App (Maybe Integer)
getMemberNumber user = do
  n ← queryValue' "SELECT member_no FROM member "
  "WHERE username = ?" [toSql user]
  return $ maybe Nothing fromSql n

getMemberName :: Integer → App (Maybe String)
getMemberName number = do
  maybe Nothing fromSql < $ > queryValue' "SELECT username FROM member "
  "WHERE member_no = ?" [toSql number]

```

To logout a member, we simply clear their auth cookie and redirect them somewhere sensible. If you want to send a client somewhere other than the front page after logout, add a `redirect` query string or POST parameter.

```

logout :: App CGIResult
logout = do
  deleteAuthCookie
  ref ← referrerOrVocabulink
  redirect ≪≪ getInputDefault ref "redirect"

```

## 11.2 New Members

To register a new member we need their desired username, a password, and optionally an email address.

```

data Registration = Registration { regUser :: String,
  regEmail :: String,
  regPass :: String }

```

They must also agree to the Terms of Use.

```

register :: AppForm Registration
register = plug (\xhtml → table << (xhtml + + + tfoot << tabularSubmit "Sign Up"))
  (reg < $ > uniqueUser
    < * > uniqueEmailAddress
    < * > passConfirmed
    < * > termsOfUse 'check' ensure (isJust)
    "You must agree to the Terms of Use.")
  where reg u e p _ = Registration u e p

```

We're very permissive with usernames. They just need to be between 3 and 32 characters long.

Since this is a site about learning languages, we really want members to be able to express themselves with their username. This may turn out to be a major pain to deal with for things like URIs, but there's 1 way to find out...

```

username :: AppForm String
username = (plug (tabularInput "Username") $ F.input Nothing) 'check' ensures
  [((≥ 3) ∘ length, "Your username must be 3 characters or longer."),
   ((≤ 32) ∘ length, "Your username must be 32 characters or shorter.")]

```

During registration, we want to make sure that the username the client is trying to register with isn't already in use.

```

uniqueUser :: AppForm String
uniqueUser = username 'checkM' ensureM valid err where
  valid user = do vs ← queryTuples' "SELECT username FROM member "
    "WHERE username ILIKE ?" [toSql user]
  return $ maybe False (≡ []) vs
  err = "That username is unavailable."

```

Our password input is as permissive as our username input.

```

passwd :: String → AppForm String
passwd l = (plug (tabularInput l) $ F.password Nothing) 'check' ensures
  [((≥ 6) ∘ length, "Your password must be 6 characters or longer."),
   ((≤ 72) ∘ length, "Your password must be 72 characters or shorter.")]

```

During registration, we want the client to confirm their password, if for no other reason than that it's common practice.

```

passConfirmed :: AppForm String
passConfirmed = fst < $ > (passwords 'check' ensure equal err) where
  passwords = (,) < $ > passwd "Password" < * > passwd "Password (confirm)"
  equal (a, b) = a ≡ b
  err = "Passwords do not match."

```

To indicate acceptance of the Terms of Use, the member checks a box. This is still a little bit awkward because the checkbox doesn't maintain its state if validation fails. I'm hoping to fix that later when I understand formlets in more depth.

```

termsOfUse :: AppForm (Maybe String)
termsOfUse = plug (\xhtml → tr << td ! [colspan 2,

```

```

                                thestyle "text-align: center"] << [
xhtml,
stringToHtml " I agree to the ",
anchor![href "/terms-of-use"] << "Terms of Use",
stringToHtml ".") (checkbox' "")

```

We don't currently do any validation on email addresses other than to check if the address is already in use. We need to check both the `member` and `member_confirmation` relations so that we catch unconfirmed email addresses as well.

```

emailAddress :: AppForm String
emailAddress = (plug (tabularInput "Email address") $ F.input Nothing) 'check'
  ensures
    [((≠ ""), "Enter an email address."),
     ((≤ 320) ∘ length, "Your email address must be "
"320 characters or shorter.")]

uniqueEmailAddress :: AppForm String
uniqueEmailAddress = emailAddress 'checkM' ensureM valid err where
  valid email = do vs ← queryTuples' "(SELECT email FROM member "
"WHERE email = ?) "
"UNION "
"(SELECT email FROM member_confirmation "
"WHERE email = ?)"
                                [toSql email, toSql email]
  return $ maybe False (≡ []) vs
  err = "That email address is unavailable."

```

The registration process consists of a single form. Once the user has registered, we log them in and redirect them to the front page.

```

registerMember :: App CGIResult
registerMember = do
  res ← runForm register $ Right noHtml
  case res of
    Left xhtml → simplePage "Sign Up for Vocabulink" [] [xhtml]
    Right reg → do
      memberNo ← quickInsertNo'
        "INSERT INTO member (username, password_hash) "
"VALUES (?, crypt(?, gen_salt('bf')))"
        [toSql (regUser reg), toSql (regPass reg)]
        "member_member_no_seq"
      case memberNo of
        Nothing → error "Registration failure (this is not your fault)."
        Just n → do
          ip ← remoteAddr
          setAuthCookie n (regUser reg) ip
          sendConfirmationEmail n reg
          redirect "/"

```

Once a user registers, they can log in. However, they won't be able to use most member-specific functions until they've confirmed their email address. This is to make sure that people cannot impersonate or spam others.

Email confirmation consists of generating a unique random string and emailing it to the member as a hyperlink. Once they click the hyperlink we consider the email address confirmed.

```

sendConfirmationEmail :: Integer → Registration → App (Maybe ())
sendConfirmationEmail memberNo r = do
  quickStmt' "INSERT INTO member_confirmation "
  "(member_no, hash, email) VALUES "
  "(?, md5(random()::text), ?)"
  [toSql memberNo, toSql (regEmail r)]
  hash ← queryValue' "SELECT hash FROM member_confirmation "
  "WHERE member_no = ?" [toSql memberNo]
  case hash of
    Nothing → return Nothing
    Just h → do
      let email = unlines [
          "Welcome to Vocabulink.",
          "",
          "Click http://www.vocabulink.com/member/confirmation/" ++
            (fromSql h) ++ " to confirm your email address."
        ]
          res ← liftIO $ sendMail (regEmail r) "Welcome to Vocabulink" email
          maybe (return Nothing) (λ_ → quickStmt'
            "UPDATE member_confirmation "
            "SET email_sent = current_timestamp "
            "WHERE member_no = ?" [toSql memberNo]) res

```

This is the place that the dispatcher will send the client to if they click the hyperlink in the email. If confirmation is successful it redirects them to some hopefully useful page.

```

confirmEmail :: String → App CGIResult
confirmEmail hash = do
  memberNo ← asks appMemberNo
  case memberNo of
    Nothing → redirect ≪≪ reversibleRedirect "/member/login"
    Just n → do
      match ← queryValue' "SELECT ? = hash FROM member_confirmation "
      "WHERE member_no = ?" [toSql hash, toSql n]
      let match' = maybe False fromSql match
          case match' of
            False → confirmEmailPage
            True → do
              res ← withTransaction' $ do
                runStmt' "UPDATE member SET email = "
                "(SELECT email FROM member_confirmation "
                "WHERE member_no = ?) "
                "WHERE member_no = ?" [toSql n, toSql n]
                runStmt' "DELETE FROM member_confirmation "
                "WHERE member_no = ?" [toSql n]

```

```

case res of
  Nothing → confirmEmailPage
  Just _ → redirect ≪≪ referrerOrVocabulink

```

This is the page we redirect unconfirmed members to when they try to interact with the site in a way that requires a confirmed email address.

```

confirmEmailPage :: App CGIResult
confirmEmailPage = do
  ref ← referrerOrVocabulink
  redirect' ← getInputDefault ref "redirect"
  support ← getSupportForm $ Just redirect'
  simplePage "Email Confirmation Required" [] [
    thediv ! [identifier "central-column"] << [
      paragraph << "In order to interact with Vocabulink, "
      "you need to confirm your email address.",
      paragraph << "If you haven't received a confirmation email "
      "or are having trouble, let us know.",
      support,
      paragraph << [
        anchor ! [href redirect'] << "Click here to go back",
        stringToHtml " to where you came from."]]]

```

At some point members (or non-members) may have difficulties with the site. I debated about giving out an email address, but I suspect that filtering for spam among support requests might be pretty difficult. I also don't want to go dead to support request email. So this uses the tried and true contact form method.

```

supportForm :: Maybe String → App (AppForm (String, String, String))
supportForm redirect' = do
  ref ← referrerOrVocabulink
  email ← asks appMemberEmail
  let redirect'' = fromMaybe ref redirect'
  emailInput = case email of
    Nothing → plug (tabularInput "Email Address") $ F.input Nothing 'check'
      ensures
        [((≠ ""), "We need an email address to contact you at.")]
    Just _ → F.hidden email
  return $ plug (λxhtml → table << [
    xhtml, tfoot << tabularSubmit "Get Support"])
  ((, ) < $ > emailInput
  < * > (plug (tabularInput "Problem") $ F.textarea Nothing) 'check' ensures
    [((≠ ""), "It would help us to know "
    "what the problem you're experiencing is ;).")]
  < * > F.hidden (Just redirect'')

```

Get a fresh support form (don't attempt to run it).

```

getSupportForm :: Maybe String → App Html
getSupportForm redirect' = do

```

```
(_, xhtml) ← runForm' ≪≪ supportForm redirect'
return $ form ! [action "/member/support", method "POST"] << xhtml
```

And finally, here is the actual support page. It's not just for member support. If the client isn't logged in it will ask for a contact email address.

Because support is so critical, in case there's an error submitting the support form we fall back to a secondary (disposable) support address.

```
memberSupport :: App CGIResult
memberSupport = do
  form' ← supportForm ≪≪ getInput "redirect"
  res ← runForm form' $ Right noHtml
  case res of
    Left xhtml → simplePage "Need Help?" []
      [theDiv ! [identifier "central-column"] << [
        paragraph ! [thestyle "text-align: center"] <<
          [stringToHtml "Have you checked the ",
            anchor ! [href "/forum/help"] << "help forum",
            stringToHtml "?"],
        xhtml]]
    Right (email, problem, redirect') → do
      supportAddress ← fromJust <$ > getOption "staticDir"
      res' ← liftIO $ sendMail supportAddress "Support Request" $
        unlines [ "Email: " ++ email,
                  "Problem: " ++ problem]
      case res' of
        Nothing → error "Error sending support request. "
        Just _ → simplePage "Support Request Sent" []
          [theDiv ! [identifier "central-column"] << [
            paragraph << "Your support request "
            "was sent successfully.",
            paragraph << [
              anchor ! [href redirect'] << "Click here to go back",
              stringToHtml " to where you came from."]]]]
```

## 12 Links

Links are the center of interest in our program. Most activities revolve around them.

```
module Vocabulink.Link (Link (.), PartialLink (.), LinkType (.),
  getPartialLink, getLinkFromPartial, getLink,
  memberLinks, latestLinks, linkPage, deleteLink,
  linksPage, linksContainingPage, newLink,
  partialLinkHtml, partialLinkFrom Values,
  drawLinkSVG, drawLinkSVG') where
```

```
import Vocabulink.App
import Vocabulink.CGI
```

```

import Vocabulink.DB
import Vocabulink.Html
import Vocabulink.Review.Html
import Vocabulink.Utils

import Data.List (partition)
import qualified Text.XHtml.Strict.Formlets as F

```

## 12.1 Link Data Types

Abstractly, a link is defined by the origin and destination lexemes it links, as well as its type. Practically, we also need to carry around information such as its link number (in the database) as well as a string representation of its type (for partially constructed links, which you'll see later).

```

data Link = Link { linkNumber      :: Integer,
                    linkTypeName   :: String,
                    linkOrigin     :: String,
                    linkOriginLang :: String,
                    linkDestination :: String,
                    linkDestinationLang :: String,
                    linkType       :: LinkType }

```

We can associate 2 lexemes in many different ways. Because different linking methods require different information, they each need different representations in the database. This leads to some additional complexity.

Each link between lexemes has a type. This type determines how the link is displayed, edited, used in statistical analysis, etc. See the Vocabulink handbook for a more in-depth description of the types.

```

data LinkType = Association | Cognate | LinkWord String String |
                 Relationship String String
deriving (Show)

```

Sometimes we need to work with a human-readable name, such as when interacting with a client or the database.

```

linkTypeNameFromType :: LinkType → String
linkTypeNameFromType Association      = "association"
linkTypeNameFromType Cognate         = "cognate"
linkTypeNameFromType (LinkWord _ _) = "link word"
linkTypeNameFromType (Relationship _ _) = "relationship"

```

Each link type also has an associated color. This makes the type of links stand out clearly in lists and graphs.

```

linkColor :: Link → String
linkColor l = case linkTypeName l of
  "association" → "#000000"
  "cognate"     → "#00AA00"

```

```

"link word"    → "#0000FF"
"relationship" → "#AA0077"
_              → "#FF00FF"

```

The link's background color is used for shading and highlighting.

```

linkBackgroundColor :: Link → String
linkBackgroundColor l = case linkTypeName l of
  "association" → "#DFDFDF"
  "cognate"     → "#DF4DF"
  "link word"   → "#DFDFFF"
  "relationship" → "#F4DFEE"
  _             → "#FFDFFF"

```

Links are created by members. Vocabulink does not own them. It merely has a license to use them (as part of the Terms of Use). So when displaying a link in full, we display a copyright notice with the member's username.

Of course, simple link types without nothing other than the origin and destination of the link do not contain copyrightable material.

```

linkCopyright :: Link → App Html
linkCopyright l = do
  case linkType l of
    LinkWord _ _ → do
      t ← queryTuple' "SELECT username, "
        "extract(year from created), "
        "extract(year from updated) "
        "FROM link, member "
        "WHERE member_no = author AND link_no = ?"
        [toSql $ linkNumber l]
      return $ paragraph! [theClass "copyright"] << stringToHtml (
        "Copyright " ++ case t of
          Just [a, c, u] → let c' = show (fromSql c :: Integer)
                               u' = show (fromSql u :: Integer)
                               r  = c' ≡ u' ? c' $ c' ++ "-" ++ u' in
                               r ++ " " ++ (fromSql a)
          _               → "unknown")
    _ → return noHtml

```

Fully loading a link from the database requires joining 2 relations. The join depends on the type of the link. But we don't always need the type-specific data associated with a link. Sometimes it's not even possible to have it, such as during interactive link construction.

We'll use a separate type to represent this. Essentially it's a link with an undefined linkType. We use a separate type to avoid passing a partial link to a function that expects a fully-instantiated link. The only danger here is writing a function that accepts a partial link and then tries to access the linkType information.

```

newtype PartialLink = PartialLink { pLink :: Link }

```

## 12.2 Storing Links

We refer to storing a link as “establishing” the link.

Each link type is expected to be potentially different enough to require its own database schema for representation. We could attempt to use PostgreSQL’s inheritance features, but I’ve decided to handle the difference between types at the Haskell layer for now. I’m actually hesitant to use separate tables for separate types as it feels like I’m breaking the relational model. However, any extra efficiency for study outranks implementation elegance (correctness?).

Establishing a link requires a member number since all links must be owned by a member.

Since we need to store the link in 2 different tables, we use a transaction. Our App-level database functions are not yet great with transactions, so we’ll have to handle the transaction manually here. You’ll also notice that some link types (such as cognates) have no additional information and hence no relation in the database.

This returns the newly established link number.

```
establishLink :: Link → Integer → App (Maybe Integer)
establishLink l memberNo = do
  r ← withTransaction' $ do
    c ← asks appDB
    linkNo ← liftIO $ insertNo c
      "INSERT INTO link (origin, destination, "
"origin_language, destination_language, "
"link_type, author) "
"VALUES (?, ?, ?, ?, ?, ?)"
    [toSql (linkOrigin l), toSql (linkDestination l),
     toSql (linkOriginLang l), toSql (linkDestinationLang l),
     toSql (linkTypeName l), toSql memberNo]
    "link_link_no_seq"
  case linkNo of
    Nothing → liftIO $ rollback c >> return Nothing
    Just n → do establishLinkType (l {linkNumber = n})
                return linkNo
  return $ fromMaybe Nothing r
```

The relation we insert additional details into depends on the type of the link and it’s easiest to use a separate function for it.

```
establishLinkType :: Link → App ()
establishLinkType l = case linkType l of
  Association → return ()
  Cognate → return ()
  (LinkWord word story) → do
    run' "INSERT INTO link_type_link_word (link_no, link_word, story) "
"VALUES (?, ?, ?)"
    [toSql (linkNumber l), toSql word, toSql story]
    return ()
  (Relationship left right) → do
    run' "INSERT INTO link_type_relationship "
"(link_no, left_side, right_side) "
"VALUES (?, ?, ?)"
```

```

    [toSql (linkNumber l), toSql left, toSql right]
return ()

```

## 12.3 Retrieving Links

Now that we've seen how we store links, let's look at retrieving them (which is slightly more complicated in order to allow for efficient retrieval of multiple links).

Retrieving a partial link is simple.

```

getPartialLink :: Integer → App (Maybe PartialLink)
getPartialLink linkNo = do
  t ← queryTuple' "SELECT link_no, link_type, origin, destination, "
"origin_language, destination_language "
"FROM link WHERE link_no = ?" [toSql linkNo]
  return $ partialLinkFromValues ≪≪ t

```

We use a helper function to convert the raw SQL tuple to a partial link value. Note that we leave the link's *linkType* undefined.

```

partialLinkFromValues :: [SqlValue] → Maybe PartialLink
partialLinkFromValues [n, t, o, d, ol, dl] = Just $
  PartialLink $ Link {linkNumber      = fromSql n,
                      linkTypeName    = fromSql t,
                      linkOrigin      = fromSql o,
                      linkDestination = fromSql d,
                      linkOriginLang  = fromSql ol,
                      linkDestinationLang = fromSql dl,
                      linkType        = ⊥ }
partialLinkFromValues _ = Nothing

```

Once we have a partial link, it's a simple matter to turn it into a full link. We just need to retrieve its type-level details from the database.

```

getLinkFromPartial :: PartialLink → App (Maybe Link)
getLinkFromPartial (PartialLink partial) = do
  linkT ← getLinkType (PartialLink partial)
  return $ (λt → Just $ partial {linkType = t}) ≪≪ linkT

```

```

getLinkType :: PartialLink → App (Maybe LinkType)
getLinkType (PartialLink p) = case p of
  (Link {linkTypeName = "association"}) → return $ Just Association
  (Link {linkTypeName = "cognate"})     → return $ Just Cognate
  (Link {linkTypeName = "link word",
         linkNumber    = n})           → do
    rs ← queryTuple' "SELECT link_word, story FROM link_type_link_word "
"WHERE link_no = ?" [toSql n]
    case rs of
      Just [linkWord, story] → return $ Just $
        LinkWord (fromSql linkWord) (fromSql story)

```

```

    _ → return Nothing
  (Link { linkTypeName = "relationship",
         linkNumber    = n }) → do
    rs ← queryTuple' "SELECT left_side, right_side "
"FROM link_type_relationship "
"WHERE link_no = ?" [toSql n]
    case rs of
      Just [left, right] → return $ Just $
        Relationship (fromSql left) (fromSql right)
      _ → return Nothing
  _ → error "Bad partial link."

```

We now have everything we need to retrieve a full link in 1 step.

```

getLink :: Integer → App (Maybe Link)
getLink linkNo = do
  l ← getPartialLink linkNo
  maybe (return Nothing) getLinkFromPartial l

```

We already know what types of links exist, but we want only the active link types (some, like Relationship, are experimental) sorted by how common they are.

```

activeLinkTypes :: [String]
activeLinkTypes = ["link word", "association", "cognate"]

```

## 12.4 Deleting Links

Links can be deleted by their owner. They're not actually removed from the database, as doing so would require removing the link from other members' review sets. Instead, we just flag the link as deleted so that it doesn't appear in most contexts.

```

deleteLink :: Integer → App CGIResult
deleteLink linkNo = do
  res ← quickStmt' "UPDATE link SET deleted = TRUE "
"WHERE link_no = ?" [toSql linkNo]
  case res of
    Nothing → error "Failed to delete link."
    Just _ → redirect ≪≪ referrerOrVocabulink

```

## 12.5 Displaying Links

Drawing links is a rather complicated process due to the limitations of HTML. Fortunately there is Raphaël (<http://raphaeljs.com/reference.html>) which makes some pretty fancy link drawing possible via JavaScript. You'll need to make sure to include both JS "raphael" and JS "link-graph" as dependencies when using this.

```

drawLinkSVG :: Link → Html
drawLinkSVG = drawLinkSVG' "drawLink"

```

```

drawLinkSVG' :: String → Link → Html
drawLinkSVG' f link = script << primHtml (
  "connect(window, 'onload', partial(" ++ f ++ ", " ++
  showLinkJSON link ++ ");") ++ +
  thediv ![identifier "graph", thestyle "height: 100px"] << noHtml

```

It seems that the JSON library author does not want us making new instances of the *JSON* class. Oh well, I didn't want to write *readJSON* anyway.

```

showLinkJSON :: Link → String
showLinkJSON link = let obj = [("orig", linkOrigin link),
  ("dest", linkDestination link),
  ("color", linkColor link),
  ("bgcolor", linkBackgroundColor link),
  ("label", linkLabel $ linkType link)] in
  encode $ toJSObject obj
  where linkLabel (LinkWord word _) = word
        linkLabel _ = ""

```

Displaying an entire link involves not just drawing a graphical representation of the link but displaying its type-level details as well.

```

displayLink :: Link → Html
displayLink l = concatHtml [
  drawLinkSVG l,
  thediv ![theclass "link-details"] << linkTypeHtml (linkType l)

```

```

linkTypeHtml :: LinkType → Html
linkTypeHtml Association = noHtml
linkTypeHtml Cognate = noHtml
linkTypeHtml (LinkWord _ story) =
  markdownToHtml story
linkTypeHtml (Relationship leftSide rightSide) =
  paragraph ![thestyle "text-align: center"] << [
    stringToHtml "as", br,
    stringToHtml $ leftSide ++ " → " ++ rightSide]

```

Sometimes we don't need to display all of a links details. This displays a partial link more compactly, such as for use in lists, etc.

```

partialLinkHtml :: PartialLink → Html
partialLinkHtml (PartialLink l) =
  anchor ![href ("/link/" ++ (show $ linkNumber l)),
    thestyle $ "color: " ++ linkColor l ++
    "; background-color: " ++ linkBackgroundColor l ++
    "; border: 1px solid " ++ linkColor l] <<
    (linkOrigin l ++ " → " ++ linkDestination l)

```

Each link gets its own URI and page. Most of the extra code in the following is for handling the display of link operations (“review”, “delete”, etc.), dealing with retrieval exceptions, etc.

```

linkPage :: Integer → App CGIResult
linkPage linkNo = do
  memberNo ← asks appMemberNo
  l ← getLink linkNo
  case l of
    Nothing → output404 ["link", show linkNo]
    Just l' → do
      review ← reviewIndicator linkNo
      owner ← queryValue' "SELECT author = ? FROM link WHERE link_no = ?"
                [toSql memberNo, toSql linkNo]
      ops ← linkOperations linkNo $ maybe False fromSql owner
      copyright ← linkCopyright l'
      let orig = linkOrigin l'
          dest = linkDestination l'
          stdPage (orig ++ " -> " ++ dest) [
            CSS "link", JS "MochiKit", JS "raphael", JS "link-graph" []
            [drawLinkSVG l',
             thediv! [theclass "link-ops"] << [review, ops],
             thediv! [theclass "link-details"] << linkTypeHtml (linkType l'),
             copyright]

```

Each link can be “operated on”. It can be reviewed (added to the member’s review set) and deleted (marked as deleted). In the future, I expect operations such as “tag”, “rate”, etc.

```

linkOperations :: Integer → Bool → App Html
linkOperations n True = do
  deleted ← queryValue' "SELECT deleted FROM link "
"WHERE link_no = ?" [toSql n]
  return $ case deleted of
    Just d → if fromSql d
      then paragraph << "Deleted"
      else form! [action ("/link/" ++ (show n) ++ "/delete"), method "POST"] <<
        submit "" "Delete"
    _ → stringToHtml
      "Can't determine whether or not link has been deleted."
linkOperations _ False = return noHtml

```

## 12.6 Finding Links

While Vocabulink is still small, it makes sense to have a page just for displaying all the (non-deleted) links in the system. This will probably go away eventually.

```

linksPage :: String → (Int → Int → App (Maybe [PartialLink])) → App CGIResult
linksPage title f = do
  (pg, n, offset) ← currentPage
  ts ← f offset (n + 1)
  case ts of
    Nothing → error "Error while retrieving links."
    Just ps → do

```

```

pagerControl ← pager pg n $ offset + (length ps)
simplePage title [CSS "link"] [
  unordList (map partialLinkHtml (take n ps))!
    [identifier "central-column", theclass "links"],
  pagerControl]

```

A more practical option for the long run is providing search. “Containing” search is a search for links that “contain” the given “focus” lexeme on one side or the other of the link. The term “containing” is a little misleading and should be changed at some point.

For now we use exact matching only as that can use an index. Fuzzy matching is going to require configuring full text search or a separate search daemon.

```

linksContainingPage :: String → App CGIResult
linksContainingPage focus = do
  ts ← queryTuples' "SELECT link_no, link_type, origin, destination, "
    "origin_language, destination_language "
    "FROM link "
    "WHERE NOT deleted "
    "AND (origin LIKE ? OR destination LIKE ?) "
    "LIMIT 20"
    [toSql focus, toSql focus]
  case ts of
    Nothing → error "Error while retrieving links."
    Just ls → simplePage ("Found " ++ (show $ length ls) ++
      " link" ++ (length ls == 1 ? "" $ "s") ++
      " containing \"" ++ focus ++ "\"")
      [CSS "link",
       JS "MochiKit", JS "raphael", JS "link-graph"]
      (linkFocusBox focus (catMaybes $ map partialLinkFromValues ls))

```

When the links containing a search term have been found, we need a way to display them. We do so by drawing a “link graph”: a circular array of links.

Before we can display the graph, we need to sort the links into “links containing the focus as the origin” and “links containing the focus as the destination”.

If you’re trying to understand this function, it helps to read the JavaScript it outputs and digest each local function separately.

```

linkFocusBox :: String → [PartialLink] → [Html]
linkFocusBox focus links = [
  script << primHtml
    ("connect(window, 'onload', partial(drawLinks," ++
     encode focus ++ "," ++
     jsonNodes ("/link?input2=" ++ focus) origs ++ "," ++
     jsonNodes ("/link?input0=" ++ focus) dests ++ "));"),
  thediv ! [identifier "graph"] << noHtml]
  where partitioned = partition ((≡ focus) ∘ linkOrigin ∘ pLink) links
        origs       = snd partitioned
        dests       = fst partitioned
        jsonNodes url xs = encode $ insertMid
          (toJSObject [("orig", "new link"),

```

```

        ("dest", "new link"),
        ("color", "#000000"),
        ("bgcolor", "#DFDFDF"),
        ("style", "dotted"),
        ("url", url))
    (map (λo → let o' = pLink o in
            toJSObject [("orig", linkOrigin o'),
                        ("dest", linkDestination o'),
                        ("color", linkColor $ pLink o),
                        ("bgcolor", linkBackgroundColor $ pLink o),
                        ("number", show $ linkNumber $ pLink o)] xs)
insertMid :: a → [a] → [a]
insertMid x xs = let (l,r) = foldr (λa~(x',y') → (a : y', x')) ([], []) xs in
                reverse l ++ [x] ++ r

```

## 12.7 Creating New Links

We want the creation of new links to be as simple as possible. For now, it's done on a single page. The form on the page dynamically updates (via JavaScript, but not AJAX) based on the type of the link being created.

This is very large because it handles generating the form, previewing the result, and dispatching the creation of the link on successful form validation.

```

newLink :: App CGIResult
newLink = withRequiredMemberNumber $ λmemberNo → do
    uri    ← requestURI
    meth   ← requestMethod
    preview ← getInput "preview"
    establishF ← establish activeLinkTypes
    (status, xhtml) ← runForm' establishF
    case preview of
        Just _ → do
            let preview' = case status of
                    Failure failures → unordList failures
                    Success link     → thediv ![theclass "preview"] <<
                        displayLink link
                simplePage "Create a Link (preview)" deps
                    [preview',
                     form ! [ thestyle "text-align: center",
                               action (uriPath uri), method "POST" ] <<
                       [xhtml, actionBar]]
            Nothing → do
                case status of
                    Failure failures → simplePage "Create a Link" deps
                        [form ! [thestyle "text-align: center",
                                   action (uriPath uri), method "POST" ] <<
                          [meth ≡ "GET" ? noHtml $ unordList failures,
                           xhtml, actionBar]]
                    Success link → do

```

```

    linkNo ← establishLink link memberNo
  case linkNo of
    Just n → redirect $ "/link/" ++ (show n)
    Nothing → error "Failed to establish link."
  where deps = [ CSS "link", JS "MochiKit", JS "link",
                JS "raphael", JS "link-graph" ]
    actionBar = thediv ![thestyle "margin-left: auto; margin-right: auto; "
"width: 12em"] <<
      [submit "preview" "Preview" !
        [thestyle "float: left; width: 5.5em"],
        submit "" "Link" ![thestyle "float: right; width: 5.5em"],
        paragraph ![thestyle "clear: both"] << noHtml]

```

Here's a form for creating a link. It gathers all of the required details (origin, destination, and link type details).

```

establish :: [String] → App (AppForm Link)
establish ts = do
  originPicker ← languagePicker $ Left ()
  destinationPicker ← languagePicker $ Right ()
  return (mkLink < $ > lexemeInput "Origin"
    < * > plug (++ +stringToHtml " ") originPicker
    < * > lexemeInput "Destination"
    < * > destinationPicker
    < * > linkTypeInput ts)

```

When creating a link from a form, the link number must be undefined until the link is established in the database. Also, because of the way formlets work (or how I'm using them), we need to retrieve the link type name from the link type.

```

mkLink :: String → String → String → String → LinkType → Link
mkLink o ol d dl t = Link { linkNumber      = ⊥,
                             linkTypeName   = linkTypeNameFromType t,
                             linkOrigin     = o,
                             linkOriginLang = ol,
                             linkDestination = d,
                             linkDestinationLang = dl,
                             linkType       = t }

```

The lexeme is the origin or destination of the link.

```

lexemeInput :: String → AppForm String
lexemeInput l = l `formLabel` F.input Nothing `check` ensures
  [((≠ "")      , l ++ " is required."),
   ((≤ 64) ∘ length, l ++ " must be 64 characters or shorter.")]

```

Each lexeme needs to be annotated with its language (to aid with disambiguation, searching, and sorting). Most members are going to be studying a single language, and it would be cruel to make them scroll through a huge list of languages each time they wanted to create a new link. So what we do is sort languages that the member has already used to the top of the list (based on frequency).

This takes an either parameter to signify whether you want origin language (Left) or destination language (Right). They are sorted separately.

```
languagePicker :: Either () () → App (AppForm String)
languagePicker side = do
  let side' = case side of
        Left _  → "origin"
        Right _ → "destination"
      memberNo ← asks appMemberNo
      langs ← (map pairUp) ∘ fromJust < $ > queryTuples'
              ("SELECT " ++ side' ++ "_language, name "
"FROM link, language "
"WHERE language.abbr = link." ++ side' ++ "_language "
"AND link.author = ? "
"GROUP BY " ++ side' ++ "_language, name "
"ORDER BY COUNT(" ++ side' ++ "_language) DESC")
              [toSql memberNo]
      allLangs ← (map pairUp) ∘ fromJust < $ > queryTuples'
                 "SELECT abbr, name FROM language ORDER BY abbr" []
  let choices = langs ++ [("", "")] ++ allLangs
  return $ F.select choices (Just $ fst ∘ head $ choices) 'check' ensures
        [((≠ ""), side' ++ " language is required")]
  where pairUp :: [SqlValue] → (String, String)
        pairUp [x, y] = (fromSql x, fromSql y)
        pairUp _      = error "Invalid pair."
```

We have a bit of a challenge with link types. We want the form to adjust dynamically using JavaScript when a member chooses one of the link types from a select list. But we also want form validation using formlets. Formlets would be rather straightforward if we were using a 2-step process (choose the link type, submit, fill in the link details, submit). But it's important to keep the link creation process simple (and hence 1-step).

The idea is to generate all the form fields for every possible link type in advance, with a default hidden state. Then JavaScript will reveal the appropriate fields when a link type is chosen. Upon submit, all link type fields for all but the selected link type will be empty (or unnecessary). When running the form, we will instantiate all of them, but then *linkTypeS* will select just the appropriate one based on the `<select>`.

The main challenge here is that we can't put the validation in the link types themselves. We have to move it into *linkTypeInput*. The problem comes from either my lack of understanding of Applicative Functors, or the fact that by the time the formlet combination strategy (Failure) runs, the unused link types have already generated failure because they have no way of knowing if they've been selected ("idioms are ignorant").

I'm deferring a proper implementation until it's absolutely necessary. Hopefully by then I will know more than I do now.

```
linkTypeInput :: [String] → AppForm LinkType
linkTypeInput ts = (linkTypeS < $ > plug (λxhtml →
                                     paragraph << [xhtml,
helpButton "/article/understanding-link-types" Nothing])
                  ("Link Type" 'formLabel' linkSelect Nothing)
                  < * > pure Association
```

```

< * > pure Cognate
< * > fieldset' "link-word" linkTypeLinkWord
< * > fieldset' "relationship" linkTypeRelationship)
'check' ensure complete
  "Please fill in all the link type fields."
where linkSelect = F.select $ zip ts ts
  complete Association      = True
  complete Cognate         = True
  complete (LinkWord w s)  = (w ≠ "") ∧ (s ≠ "")
  complete (Relationship l r) = (l ≠ "") ∧ (r ≠ "")
  fieldset' ident          = plug
    (fieldset! [identifier ident, thestyle "display: none"]<<)

linkTypeS :: String → LinkType → LinkType → LinkType → LinkType → LinkType
linkTypeS "association" l _ _ _ = l
linkTypeS "cognate"      _ l _ _ = l
linkTypeS "link word"   _ _ l _ = l
linkTypeS "relationship" _ _ _ l = l
linkTypeS _ _ _ _ _ = error "Unknown link type."

linkTypeLinkWord :: AppForm LinkType
linkTypeLinkWord = LinkWord < $ > "Link Word" 'formLabel' F.input Nothing
  < * > F.textarea (Just "Write a story linking the 2 words here.")

linkTypeRelationship :: AppForm LinkType
linkTypeRelationship = Relationship < $ >
  plug (+++stringToHtml " is to ") (F.input Nothing) < * > F.input Nothing

```

We want to be able to display links in various ways. It would be really nice to get lazy lists from the database. However, lazy JDBC results don't seem to work too well in my experience (at least not with PostgreSQL). For now, you need to specify how many results you want, as well as an offset.

Here we retrieve multiple links at once. This was the original motivation for dividing link types into full and partial. Often we need to retrieve links for simple display but we don't need or want extra trips to the database. Here we need only 1 query instead of potentially `limit` queries.

We don't want to display deleted links (which are left in the database for people still reviewing them). There is some duplication of SQL here, but I have yet found a nice way to generalize these functions.

The first way to retrieve links is to just grab all of them, starting at the most recent. This assumes the ordering of links is determined by link number.

```

latestLinks :: Int → Int → App (Maybe [PartialLink])
latestLinks offset limit = do
  ts ← queryTuples' "SELECT link_no, link_type, origin, destination, "
  "origin_language, destination_language "
  "FROM link WHERE NOT deleted "
  "ORDER BY link_no DESC "

```

```
"OFFSET ? LIMIT ?" [toSql offset, toSql limit]
  return $ (catMaybes ◦ map partialLinkFromValues) 'liftM' ts
```

Another way we retrieve links is by author (member). These just happen to be sorted by link number as well.

```
memberLinks :: Integer → Int → Int → App (Maybe [PartialLink])
memberLinks memberNo offset limit = do
  ts ← queryTuples' "SELECT link_no, link_type, origin, destination, "
"origin_language, destination_language "
"FROM link "
"WHERE NOT deleted AND author = ? "
"ORDER BY link_no DESC "
"OFFSET ? LIMIT ?"
  [toSql memberNo, toSql offset, toSql limit]
  return $ (catMaybes ◦ map partialLinkFromValues) 'liftM' ts
```

## 13 Review

Creating links is great, but viewing a link doesn't mean that you've learned it. We need a way to present links to members for regular (scheduled) reviews.

```
module Vocabulink.Review (newReview, linkReviewed, nextReview) where
```

For now, we have only 1 review algorithm (SuperMemo 2).

```
import qualified Vocabulink.Review.SM2 as SM2
```

```
import Vocabulink.App
import Vocabulink.CGI
import Vocabulink.DB
import Vocabulink.Html
import Vocabulink.Link
```

### 13.1 Review Scheduling

When a member indicates that they want to review a link, we just add it to the `link_to_review` relation. This may change if we ever support multiple review sets. The link review time is set to the current time by default so that it immediately shows up for review (something we want no matter which algorithm we're using).

We need to give the user feedback that they've successfully added the link to their review set. For now, we redirect them back to the referrer because we assume it will be the link page (which will then indicate in some way that they're reviewing the link now). However, this is a good candidate for an asynchronous JavaScript call.

```
newReview :: Integer → Integer → App CGIResult
newReview memberNo linkNo = do
  res ← quickStmt' "INSERT INTO link_to_review (member_no, link_no) "
```

```

"VALUES (?, ?)" [toSql memberNo, toSql linkNo]
case res of
  Nothing → error "Error scheduling link for review."
  Just _ → redirect ≪≪ referrerOrVocabulink

```

The client indicates a completed review with a POST to `/review/linknumber` which will be dispatched to `linkReviewed`. Once we schedule the next review time for the link, we move on to the next in their set.

```

linkReviewed :: Integer → Integer → App CGIResult
linkReviewed memberNo linkNo = do
  recall      ← readRequiredInput "recall"
  recallTime ← readRequiredInput "recall-time"
  res ← scheduleNextReview memberNo linkNo recall recallTime
case res of
  Nothing → error "Failed to schedule next review."
  Just _ → redirect "/review/next"

```

We need to schedule the next review based on the review algorithm in use. The algorithm needs to know how well the item was remembered. Also, we log the amount of time it took to recall the item. The SM2 algorithm does not use this information (nor any SuperMemo algorithm that I know of), but we may find it useful when analyzing data later.

`recall` is passed as a real number between 0 and 1 to allow for future variations in recall rating (such as fewer choices than 1 through 5 or less discrete options like a slider). 0 indicates complete failure while 1 indicates perfect recall.

`previous` is passed as well. This is the time in seconds between this and the last review (the actual time difference, not the scheduled time difference).

All database updates during this process are wrapped in a transaction.

```

scheduleNextReview :: Integer → Integer → Double → Integer → App (Maybe ())
scheduleNextReview memberNo linkNo recall recallTime = do
  previous ← previousInterval memberNo linkNo
case previous of
  Nothing → return Nothing
  Just p → withTransaction' $ do
    seconds ← SM2.reviewInterval memberNo linkNo p recall
case seconds of
  Nothing → throwDyn ()
  Just s → do
    run' "INSERT INTO link_review (member_no, link_no, recall, "
    "recall_time, target_time) "
    "VALUES (?, ?, ?, ?, "
    "(SELECT target_time FROM link_to_review "
    "WHERE member_no = ? AND link_no = ?))"
    [toSql memberNo, toSql linkNo, toSql recall,
     toSql recallTime, toSql memberNo, toSql linkNo]
    run' ("UPDATE link_to_review "
    "SET target_time = current_timestamp + interval "
    "' ' + (show s) + " seconds" + "' "
    "WHERE member_no = ? AND link_no = ?")

```

```
[toSql memberNo, toSql linkNo]
  return ()
```

## 13.2 Review Pages

Here's the entry point for the client to request reviews. It's pretty simple: we just request the next link from `link_to_review` by `target_time`. If there's none, we send the member to a "congratulations" page. If there is a link for review, we send them to the review page.

```
nextReview :: Integer → App CGIResult
nextReview memberNo = do
  linkNo ← queryTuples'
    "SELECT link_no FROM link_to_review "
    "WHERE member_no = ? AND current_timestamp >= target_time "
    "ORDER BY target_time ASC LIMIT 1" [toSql memberNo]
  case linkNo of
    Just [] → noLinksToReviewPage
    Just [[n]] → reviewLinkPage $ fromSql n
    _ → error "Failed to retrieve next link for review."
```

The review page is pretty simple. It displays a link without the typical link-type decorations and with the destination node covered by a question mark. Once the member clicks the question mark or presses the space bar (to find out what is hidden beneath it) it reveals the lexeme, records the total amount of recall time taken, and displays a recall feedback form (currently a row of 6 buttons for working with the SM2 algorithm).

Once the member clicks a recall number, it sends the information off to `linkReviewed` to record the details and schedule the next review. This sends the client to `nextReview` which begins the process all over again.

```
reviewLinkPage :: Integer → App CGIResult
reviewLinkPage linkNo = do
  l ← getLink linkNo
  case l of
    Nothing → simplePage "Error: Unable to retrieve link." [CSS "link" []]
    Just l' → do
      let source = linkOrigin l
          dest = linkDestination l
      stdPage ("Review: " ++ source ++ " - ?")
        [CSS "link", JS "MochiKit", JS "review",
         JS "raphael", JS "link-graph" []]
        [drawLinkSVG' "drawReview" l',
         form! [action ("/review/" ++ (show linkNo)), method "POST"] <<
           [hidden "recall-time" "",
            hidden "hidden-lexeme" dest,
            fieldset! [identifier "recall-buttons", thestyle "display: none"] <<
              map (recallButton 5) [0..5]]]
```

This creates a "recall button". It returns a button with a decimal recall value based on an integral button number. It hopefully allows us to make the recall options more flexible in the future.

You may get unpleasant results when passing a *total* that doesn't cleanly divide *i*.

```
recallButton :: Integer → Integer → Html
recallButton total i = let q :: Double = (fromIntegral i) / (fromIntegral total) in
  button ! [name "recall", value (show q)] << show i
```

When a member has no more links to review for now, let's display a page letting them know that.

Here's a critical chance to:

- Give positive feedback to encourage the behavior of getting through the review set.
- Point the member to other places of interest on the site.

But for now, the page is pretty plain.

```
noLinksToReviewPage :: App CGIResult
noLinksToReviewPage = do
  simplePage "No Links to Review" [CSS "link"] [
    thediv ! [identifier "central-column"] << [
      paragraph << "Take a break! "
      "You don't have any links to review right now."]]
```

In order to determine the next review interval, the review scheduling algorithm may need to know how long the last review period was (in fact, any algorithm based on spaced repetition will). This returns the actual, not scheduled, amount of time between the current and last review in seconds.

Note that this will not work before the link has been reviewed. We expect that the review algorithm does not have to be used for determining the first review (immediate).

```
previousInterval :: Integer → Integer → App (Maybe Integer)
previousInterval memberNo linkNo = do
  v ← queryValue' "SELECT COALESCE(extract(epoch from "
"current_timestamp - "
"(SELECT actual_time FROM link_review "
"WHERE member_no = ? AND link_no = ? "
"ORDER BY actual_time DESC LIMIT 1)), "
"extract(epoch from current_timestamp - "
"(SELECT target_time FROM link_to_review "
"WHERE member_no = ? AND link_no = ?)))"
    [toSql memberNo, toSql linkNo,
     toSql memberNo, toSql linkNo]
  return $ fmap fromSql v
```

## 14 Review Html

This is separate from the main Review module only to break cyclical imports. It may end up being more work than it's worth.

```
module Vocabulink.Review.Html (reviewBox, reviewIndicator) where
```

```

import Vocabulink.App
import Vocabulink.DB
import Vocabulink.Utils

```

```

import Text.XHtml.Strict

```

We display the number of links that are waiting for review for logged in members in the standard page header. Reviewing is currently the primary function of Vocabulink, and we want it prominently displayed.

The idea is that a member will go to the site, and we want them to be instantly reminded that they have links to review. Or, if a link for review becomes due while they are browsing another part of the site, we want them to be notified.

```

reviewBox :: App Html
reviewBox = withMemberNumber noHtml $ \memberNo → do
  n ← numLinksToReview memberNo
  return $ case n of
    Just 0 → anchor ![href "/links", theclass "review-box",
                      thestyle "color: black"] <<
              "No links to review"
    Just n' → anchor ![href "/review/next", theclass "review-box"] <<
              [
                strong << (show n'),
                stringToHtml (n' > 1 ? " links" $ " link"),
                stringToHtml " to review"
              ]
    Nothing → stringToHtml "Error finding links for review."

```

This retrieves the number of links that a user has for review right now.

```

numLinksToReview :: Integer → App (Maybe Integer)
numLinksToReview memberNo = do
  v ← queryValue' "SELECT COUNT(*) FROM link_to_review "
  "WHERE member_no = ? AND current_timestamp > target_time"
  [toSql memberNo]
  return $ fmap fromSql v

```

When displaying a link, it's useful to show its review status (or a method to add it to a review set if it's not). This returns an Html element that will do both based on the link number and the currently logged in member.

```

reviewIndicator :: Integer → App (Html)
reviewIndicator linkNo = do
  memberNo ← asks appMemberNo
  case memberNo of
    Nothing → return $ paragraph ![theclass "review-box login"] <<
              anchor ![href "/member/login"] << "Login to Review"
    Just n → do
      r ← reviewing n linkNo
      case r of
        Nothing → return $ paragraph ![theclass "review-box"] <<
                  "Unable to determine review status."

```

```

Just r' → return $ r' ? paragraph ! [theclass "review-box reviewing"] <<
  "Reviewing" $
  form ! [action ("/review/" ++ (show linkNo) ++ "/add"),
    method "POST", theclass "review-box review"] <<
  [submit "review" "Review"]

```

*reviewing* determines whether or not a member is already reviewing a link. This will be true only if the member is currently reviewing the link, not if they've reviewed it in the past and later removed it from their review set.

```

reviewing :: Integer → Integer → App (Maybe Bool)
reviewing memberNo linkNo =
  (≠ []) < $$ > queryTuples' "SELECT link_no FROM link_to_review "
  "WHERE member_no = ? AND link_no = ?"
  [toSql memberNo, toSql linkNo]

```

## 15 Review Algorithm SM2

This is SuperMemo algorithm SM-2 (<http://www.supermemo.com/english/ol/sm2.htm>)

```

module Vocabulink.Review.SM2 (reviewInterval) where

```

```

import Vocabulink.App
import Vocabulink.DB

```

Each algorithm needs to export a *reviewInterval* function that accepts some basic information about a completed review and returns the time (in seconds from now) to schedule the next review for. *reviewInterval* will be called with an active transaction, so it must not commit anything to the database (such as using `quickStmnt`, etc.).

This function is responsible for doing any updates to the database that it needs in order to accurately return the next review interval when that time comes.

This should return `Nothing` if there was an error or `0` if the item needs to be repeated immediately.

```

reviewInterval :: Integer → Integer → Integer → Double → App (Maybe Integer)
reviewInterval memberNo linkNo previous recall = do
  let p = daysFromSeconds previous
      q :: Integer = round $ recall * 5 -- The algorithm expects 0-5, not 0-1.
      stats ← queryTuples' "SELECT n, EF FROM link_sm2 "
  "WHERE member_no = ? AND link_no = ?"
  [toSql memberNo, toSql linkNo]

  case stats of
    Just [] → if q < 3 -- This item is not yet learned.
      then return $ Just 0
      else do let n = 1
              ef = easinessFactor' 2.5 q
              createSM2 memberNo linkNo n ef
              return $ Just $

```

```

                                secondsFromDays (interval p n ef)
Just [[n', ef']] → let ef = fromSql ef'
                    n :: Integer = fromSql n' in
                    if q < 3 -- We need to restart the learning process.
                    then do updateSM2 memberNo linkNo 1 ef
                           return $ Just 0
                    else do let newEF = easinessFactor' ef q
                               n''    = n + 1
                           updateSM2 memberNo linkNo n'' newEF
                           return $ Just $
                                secondsFromDays (interval p n'' newEF)
                    → return Nothing

```

The algorithm works with days, but we keep track of seconds.

```

daysFromSeconds :: Integer → Double
daysFromSeconds s = (fromIntegral s) / (24 * 60 * 60)

```

```

secondsFromDays :: Double → Integer
secondsFromDays d = round $ d * 24 * 60 * 60

```

The following are based on a direct translation of the published algorithm. The variable names are taken directly from the document.

The first 2 review intervals are fixed. The following are based on the duration of the previous interval ( $p$ ) and the easiness factor ( $ef$ ).

```

interval :: Double → Integer → Double → Double
interval _ 1 _ = 1.0
interval _ 2 _ = 6.0
interval p _ ef = p * ef

```

The easiness factor is the core calculation of the SM-2 algorithm.

We have a unique advantage in that it's possible for us to determine the EF (easiness factor) of an item through collaboration. However, I'm not sure if that's necessary or will pay off as much as moving to a newer algorithm. But for now we'll stick to a separate EF for each member.

```

easinessFactor' :: Double → Integer → Double
easinessFactor' ef q = max 1.3 $ ef + (0.1 - x * (0.08 + x * 0.02))
  where x :: Double = fromIntegral $ 5 - q

```

For the SM-2 algorithm to work, we need to keep track of a couple variables for each link. This establishes the variable record in the database the first time a link is reviewed.

```

createSM2 :: Integer → Integer → Integer → Double → App ()
createSM2 memberNo linkNo n ef = do
  run' "INSERT INTO link_sm2 (member_no, link_no, n, EF) "
"VALUES (?, ?, ?, ?)"
    [toSql memberNo, toSql linkNo, toSql n, toSql ef]
  return ()

```

When a link is already being reviewed, this updates the SM2 variables.

```

updateSM2 :: Integer → Integer → Integer → Double → App ()
updateSM2 memberNo linkNo n ef = do
  run' "UPDATE link_sm2 SET n = ?, EF = ? "
  "WHERE member_no = ? AND link_no = ?"
  [toSql n, toSql ef, toSql memberNo, toSql linkNo]
  return ()

```

## 16 Articles

All of Vocabulink's `www` subdomain is served by this program. As such, if we want to publish static data there, we need some outlet for it. The only form of static page we currently display is an article.

```

module Vocabulink.Article (articlePage, articlesPage, refreshArticles,
                           getArticle, articleBody, getArticles,
                           articleLinkHtml) where

import Vocabulink.App
import Vocabulink.CGI
import Vocabulink.DB
import Vocabulink.Html
import Vocabulink.Utils hiding ((< $$ >))

import Control.Monad (filterM)
import Data.Time.Format (parseTime)
import System.Directory (getDirectoryContents, getPermissions, doesFileExist,
                          readable)
import System.FilePath (takeExtension, replaceExtension, takeBaseName)
import qualified System.IO.UTF8 as IO.UTF8
import System.Locale (defaultTimeLocale)
import qualified Text.ParserCombinators.Parsec as P
import Text.ParserCombinators.Parsec.Perm ((< $$ >), (< || >), (< |? >), permute)

```

All articles have some common metadata.

The article file path is not strictly necessary to store, but it does allow an article to have a different representation on the filesystem than some translation of its title. This avoids potential problems with automatic title-to-path translation, such as characters that the filesystem doesn't like or articles with very long titles. The filepath however is just the relative filename in the articles directory, not an absolute path.

```

data Article = Article { articleFilename    :: FilePath,
                        articleAuthor      :: Maybe String,
                        articlePublishTime :: UTCTime,
                        articleUpdateTime  :: Maybe UTCTime,
                        articleSection     :: Maybe String,
                        articleTitle       :: String }

```

For now, article storage is very simple. That may need to change in the future for efficiency and to allow people without access to the article repository to contribute.

## 16.1 Retrieving Articles

Creating articles is done outside of this program. They are currently generated from Muse-mode files (<http://mwolson.org/projects/EmacsMuse.html>) using Emacs. However, we can reconstruct the metadata for an article by parsing the file ourselves.

Keeping the body of the article outside of the database gives us some of the advantages of the filesystem such as revision control.

```
articleDir :: App String
articleDir = fromJust < $ > getOption "articledir"
```

We need a way of keeping the database consistent with the filesystem. This is it. For a logged in administrator, the articles page displays a “Refresh from Filesystem” button that POSTs here.

```
refreshArticles :: App CGIResult
refreshArticles = do
  articles ← publishedArticles
  c ← asks appDB
  insert ← liftIO $ prepare c
    "INSERT INTO article (author, publish_time, update_time, "
  "section, title, filename) "
  "VALUES ((SELECT member_no FROM member "
  "WHERE username = ?), ?, ?, ?, ?, ?)"
  liftIO $ withTransaction c (\_ →
    mapM_ (\a → execute insert (rec a)) articles)
  redirect ≪≪ referrerOrVocabulink
  where rec a = [toSql $ articleAuthor      a,
                 toSql $ articlePublishTime a,
                 toSql $ articleUpdateTime a,
                 toSql $ articleSection     a,
                 toSql $ articleTitle      a,
                 toSql $ articleFilename   a]
```

Articles are created and updated via the filesystem, but we need to have the metadata on articles available in the database before we can display them via the web. One option is to actually submit articles through the web. But I consider that to be needlessly complex for now. Instead, we can just look over a directory of articles, parse the header of each for metadata, and then update the database.

We list the contents of the given directory (non-recursively) and figure out which files are published articles. Finally, we call *getArticle* for each of them. Then we return a list of Articles with undefined numbers and bodies.

```
publishedArticles :: App [Article]
publishedArticles = do
  dir ← articleDir
  ls ← liftIO $ getDirectoryContents dir
  let fullPaths = map (\l → dir ++ "/" ++ l) ls
      paths ← liftIO $ (map takeBaseName) < $ > filterM isPublished fullPaths
      catMaybes < $ > mapM articleFromFile paths
```

We consider an article (file) published if it:

1. ends with `.muse`
2. is readable
3. has a corresponding readable `.html` file

```
isPublished :: FilePath → IO Bool
isPublished f = do
  if takeExtension f ≡ ".muse"
  then do
    r1 ← isReadable f
    r2 ← isReadable $ replaceExtension f ".html"
    return $ r1 ∧ r2
  else return False
```

```
isReadable :: FilePath → IO Bool
isReadable f = do
  exists' ← doesFileExist f
  if exists'
  then do
    perms ← getPermissions f
    return $ readable perms
  else return False
```

To retrieve an article from the filesystem we just need the path to the `.muse` file. We don't generate the article HTML (that's done by Emacs), so we read that separately from a corresponding `.html` file. Both files must exist for this to succeed.

```
articleFromFile :: String → App (Maybe Article)
articleFromFile path = do
  dir ← articleDir
  muse ← liftIO $ IO.UTF8.readFile $ dir ++ "/" ++ path ++ ".muse"
  case P.parse articleHeader "" muse of
    Left e → logApp "parse error" (show e) >> return Nothing
    Right hdr → return $ Just $ hdr { articleFilename = path }
```

Each article is expected to have a particular structure. The structure is based off of a required subset of the Muse-mode directives.

An accepted article's first lines will consist of something like:

```
#title Why Learn with Vocabulink?
#author jekor
#date 2009-01-11 16:04 -0800
#update 2009-02-28 14:55 -0800
#section main
```

`#title` is a freeform title.

`#author` must be a username from the members table.

`#date` is the date in ISO-8601 format (with spaces between the date, time, and timezone).

`#update` is an optional update time (in the same format as `#date`).

`#section` is the section of the site in to publish the article. For static content “articles” such as the privacy policy, don’t include a section. For now, only “main” is supported with our simple publishing system.

Parsing an article’s metadata is pretty simple with Parsec’s permutation combinators.

We’re going to go ahead and use `fromJust` here because we don’t care about how we’re notified of errors (publishing articles is not (yet) member-facing).

```
articleHeader :: P.Parser Article
articleHeader = permute
  (mkArticle < $$ > (museDirective "title")
    < |? > (Nothing, museDir "author" >> authorP)
    < || > (museDir "date" >> dateTimeP)
    < |? > (Nothing, museDir "update" >> dateTimeP)
    < |? > (Nothing, Just < $ > museDirective "section"))
  where mkArticle title author date update section =
        Article { articleFilename    = ⊥,
                  articleAuthor      = author,
                  articlePublishTime = fromJust date,
                  articleUpdateTime  = update,
                  articleSection      = section,
                  articleTitle       = title }
```

A muse directive looks sort of like a C preprocessor directive.

```
museDirective :: String → P.Parser String
museDirective dir = museDir dir >> P.manyTill P.anyChar P.newline
```

```
museDir :: String → P.Parser ()
museDir dir = P.try (P.string ("#" ++ dir)) >> P.spaces
```

```
authorP :: P.Parser (Maybe String)
authorP = Just < $ > P.manyTill P.anyChar P.newline
```

```
dateTimeP :: P.Parser (Maybe UTCTime)
dateTimeP = parseTime defaultTimeLocale "%F %T %z" < $ >
  P.manyTill P.anyChar P.newline
```

We don’t need the article body until we go to actually display the article, so there’s no point in storing it in the article record.

```
articleBody :: Article → App Html
articleBody article = do
  path ← (+"/" ++ (articleFilename article) ++ ".html") < $ > articleDir
  liftIO $ primHtml < $ > IO.UTF8.readFile path
```

## 16.2 Retrieving Articles

To retrieve an article with need its (relative) filename (or path, or whatever you want to call it).

```
getArticle :: String → App (Maybe Article)
getArticle filename = do
  (≫=articleFromTuple) < $ >
    queryTuple' "SELECT filename, author, publish_time, "
    "section, update_time, title "
    "FROM article WHERE filename = ?" [toSql filename]
```

As with links, we use a helper function to convert a raw SQL tuple.

```
articleFromTuple :: [SqlValue] → Maybe Article
articleFromTuple [f, a, p, u, s, t] = Just $
  Article { articleFilename    = fromSql f,
           articleAuthor      = fromSql a,
           articlePublishTime = fromSql p,
           articleUpdateTime  = fromSql u,
           articleSection     = fromSql s,
           articleTitle       = fromSql t }
articleFromTuple _ = Nothing
```

To get a list of articles for display on the main articles page, we look for published articles in the database that are in the “main” section.

```
getArticles :: App (Maybe [Article])
getArticles = do
  rs ← queryTuples' "SELECT filename, author, publish_time, "
  "update_time, section, title "
  "FROM article "
  "WHERE publish_time < CURRENT_TIMESTAMP "
  "AND section = 'main' "
  "ORDER BY publish_time DESC" []
  case rs of
    Nothing → return Nothing
    Just rs' → return $ Just $ catMaybes $ map articleFromTuple rs'
```

## 16.3 Article Pages

Here’s how to display an article to the client. We don’t check here to see whether or not it’s published (past its publication date) as unpublished articles presumably don’t have links to them.

```
articlePage :: String → App CGIResult
articlePage path = do
  article ← getArticle path
  case article of
    Nothing → output404 ["article", path]
```

```

Just a → do
  body ← articleBody a
  stdPage (articleTitle a) [CSS "article"] []
  [thediv ! [theclass "article"] << body]

```

This page is for displaying a listing of published articles.

```

articlesPage :: App CGIResult
articlesPage = do
  articles ← getArticles
  memberName ← asks appMemberName
  case articles of
    Nothing → error "Error retrieving articles"
    Just as → simplePage "Articles" [CSS "article"]
      [thediv ! [theclass "article"] << [
        unordList $ map articleLinkHtml as,
        refresh memberName]]
  where refresh member = case member of
    Just "jekor" →
      form ! [action "/articles", method "POST"] <<
        submit "" "Refresh from filesystem."
    _ → noHtml

```

Create a clickable link HTML fragment for an article.

```

articleLinkHtml :: Article → Html
articleLinkHtml a = anchor ! [href ("/article/" ++ articleFilename a)] <<
  articleTitle a

```

## 17 Forum

Vocabulink uses a custom forums implementation. I created it for a couple reasons:

1. We want maximum integration and control as the forums are an integral part of the site, not just an afterthought (they are here to direct the evolution of the site).
2. Much of the forum logic is based on comments, which is common to the forums, articles, and most importantly, links. Once we have comment threads, the forums come almost for free. (Note that comments on articles and links are currently supported but will be soon.)
3. We reduce a source of security problems, and potentially maintenance problems as well. (Most software is implemented in PHP which is notorious for security problems.)

```

module Vocabulink.Forum (forumsPage, forumPage, createForum,
  newTopicPage, forumTopicPage,
  replyToForumComment, commentPreview) where

```

```

import Vocabulink.App
import Vocabulink.CGI

```

```

import Vocabulink.DB
import Vocabulink.Html
import Vocabulink.Utils

```

```

import qualified Data.ByteString.Lazy as BS
import Network.Gravatar (gravatarWith, size)
import qualified Text.XHtml.Strict.Formlets as F

```

The Forums page is a high-level look into Vocabulink's forums. Each forum is part of a group of similar forums. We have a placeholder for an eventual search function which will search through the text of all comments in all forums. Also, we include an administrative interface for creating new groups.

```

forumsPage :: App CGIResult
forumsPage = do
  res ← runForm ("Forum Group Name" `formLabel` F.input Nothing) $ Left "Create"
  memberName ← asks appMemberName
  case (res, memberName) of
    (Right s, Just "jekor") → createForumGroup s
    (Left xhtml, _) → do
      groups ← queryAttribute' "SELECT group_name FROM forum_group "
"ORDER BY position ASC" []
      groups' ← case groups of
        Just gs → do
          gs' ← mapM (renderForumGroup ∘ fromSql) gs
          return $ concatHtml gs'
        Nothing → return $ stringToHtml "Error retrieving forums."
      simplePage "Forums" forumDeps
      [ -- Forum search disabled until it works.
        -- form ! [action "/forum/threads", method "POST"] <<
        -- [ textfield "containing", submit "" "Search" ],
        groups',
        if memberName ≡ Just "jekor"
          then button ! [theClass "reveal forum-group-creator"] <<
            "New Forum Group" + + +
            theDiv ! [identifier "forum-group-creator",
              theClass "forum-group",
              theStyle "display: none"] << xhtml
          else noHtml]
      - → outputError 403 "Access Denied" []

```

The dependencies for forum pages are all the same.

```

forumDeps :: [Dependency]
forumDeps = [CSS "forum", JS "MochiKit", JS "forum", JS "form"]

```

Displaying an individual group of forums is a little bit tougher than it would seem (we have to also support the administrative interface for creating new forums within the group).

```

renderForumGroup :: String → App Html
renderForumGroup g = do

```

```

memberName ← asks appMemberName
forums ← queryTuples' "SELECT name, title, icon_filename FROM forum "
"WHERE group_name = ? ORDER BY position ASC"
      [toSql g]

case forums of
  Nothing → return $ paragraph << "Error retrieving forums"
  Just fs → do
    let fs' = map renderForum fs
        creator = case memberName of
          Just "jekor" → [button! [theClass "reveal forum-creator"] <<
                          "New Forum" ++ forumCreator]
          _             → []
        (left, right) = foldr (λa~(x, y) → (a : y, x)) ([], []) (fs' ++ creator)
    return $ thediv! [theClass "forum-group"] <<
      [h2 << g,
       unordList left! [theClass "first"],
       unordList right,
       paragraph! [theClass "clear"] << noHtml]
    where forumCreator =
      form! [identifier "forum-creator",
            thestyle "display: none",
            action "/forum/new",
            method "POST",
            enctype "multipart/form-data"] <<
        fieldset <<
          [legend << "New Forum",
           hidden "forum-group" g,
           table <<
             [tabularInput "Title" $ textField "forum-title",
              tabularInput "Icon (64x64)" $ afile "forum-icon",
              tabularSubmit "Create"]]

```

Displaying individual forums within the group is easier. Each forum requires an icon. This allows faster visual navigation to the forum of interest when first arriving at the top-level forums page.

```

renderForum :: [SqlValue] → Html
renderForum [n, t, i] =
  anchor! [href $ "/forum/" ++ (fromSql n)] << [
    image! [width "64", height "64",
           src ("http://s.vocabulink.com/icons/" ++ fromSql i)],
    stringToHtml $ fromSql t]
renderForum _ = stringToHtml "Error retrieving forum."

```

Creating a forum group is a rare administrative action. For now, we're not concerned with error handling and such. Maybe later when bringing on volunteer moderators or someone else I'll be motivated to make this nicer.

```

createForumGroup :: String → App CGIResult
createForumGroup s = do

```

```

    c ← asks appDB
    liftIO $ quickStmt c "INSERT INTO forum_group (group_name, position) "
"VALUES (?, COALESCE((SELECT MAX(position) "
"FROM forum_group), 0) + 1)"
                                [toSql s]
    liftIO memcacheFlush
    redirect ≪≪ referrerOrVocabulink

```

For now, we just upload to the icons directory in our static directory.

```

createForum :: App CGIResult
createForum = do
    icons ← iconDir
    filename ← getInputFilename "forum-icon"
    group ← getInput "forum-group"
    title ← getInput "forum-title"
    case (filename, group, title) of
        (Just f, Just g, Just t) → do
            icon ← fromJust < $ > getInputFPS "forum-icon"
            let iconfile = icons ++ "/" ++ basename f
                liftIO $ BS.writeFile iconfile icon
            c ← asks appDB
            liftIO $ quickStmt c "INSERT INTO forum (name, title, group_name, "
"position, icon_filename) "
"VALUES (?, ?, ?, "
"COALESCE((SELECT MAX(position) "
"FROM forum "
"WHERE group_name = ?), 0) + 1, ?)"
                                [toSql $ urlify t, toSql t, toSql g,
                                toSql g, toSql $ basename f]
            liftIO memcacheFlush
            redirect ≪≪ referrerOrVocabulink
        _ → error "Please fill in the form completely. "
"Make sure to include an icon."

```

The directory for forum icons is located within the static directory (and hence served from the static subdomain).

```

iconDir :: App String
iconDir = (+" /icons") ◦ fromJust < $ > getOption "staticdir"

```

Each forum page is a table of topics. Each topic lists the title (a description of the topic), how many replies the topic has received, who created the topic, and the time of the latest topic.

```

forumPage :: String → App CGIResult
forumPage forum = do
    title ← queryValue' "SELECT title FROM forum WHERE name = ?"
                                [toSql forum]
    case title of
        Nothing → output404 ["forum", forum]
        Just t → do

```

```

let tc = concatHtml $ [
  td ![theClass "topic"] <<
    form ![action (forum ++ "/new"), method "GET"]
      << submit "" "New Topic",
  td << noHtml, td << noHtml, td << noHtml]
topics ← forumTopicRows forum
stdPage ("Forum - " ++ forum) forumDeps []
[breadcrumbs [anchor ![href "../forums"] << "Forums",
  stringToHtml $ fromSql t],
thead ![identifier "topics"] << table << [
  th << tr << [
    th ![theClass "topic"] << "Topic",
    th ![theClass "replies"] << "Replies",
    th ![theClass "author"] << "Author",
    th ![theClass "last"] << "Last Comment"],
tbody << tableRows (tc : topics),
tfoot << tr << noHtml]]

```

This returns a list of forum topics, sorted by creation date, as HTML rows. It saves us some effort by avoiding any intermediate representation which we don't yet need.

The structure of the comment relations make sorting by latest comment a little bit more difficult and a task for later.

```

forumTopicRows :: String → App [Html]
forumTopicRows t = do
  topics ← queryTuples'
    "SELECT t.topic_no, t.title, t.num_replies, "
    "m1.username, c2.time, m2.username "
    "FROM forum_topic t, comment c1, comment c2, member m1, member m2 "
    "WHERE c1.comment_no = t.root_comment "
    "AND c2.comment_no = t.last_comment "
    "AND m1.member_no = c1.author "
    "AND m2.member_no = c2.author "
    "AND forum_name = ? "
    "ORDER BY t.topic_no DESC" [toSql t]
  case topics of
    Nothing → return [td << "Error retrieving topics."]
    Just ts → return $ map topicRow ts
  where topicRow :: [SqlValue] → Html
        topicRow [tn, tt, nr, ta, lt, la] =
          let tn' :: Integer = fromSql tn
              tt' :: String = fromSql tt
              nr' :: Integer = fromSql nr
              ta' :: String = fromSql ta
              lt' :: UTCTime = fromSql lt
              la' :: String = fromSql la in
            concatHtml [
              td ![theClass "topic"] <<
                anchor ![href (t ++ "/" ++ (show tn'))] << tt',
              td ![theClass "replies"] << show nr',

```

```

    td ![theclass "author"] << ta',
    td ![theclass "last"] << [
        stringToHtml $ formatSimpleTime lt',
        br,
        stringToHtml la']]
    topicRow _ = td << "Error retrieving topic."

```

The page for creating a new topic is very plain. In fact, it doesn't even have a way to preview the text of the first comment to the topic.

```

newTopicPage :: String → App CGIResult
newTopicPage n = withRequiredMemberNumber $ λ_ → do
    email ← asks appMemberEmail
    memberName ← fromJust < $ > asks appMemberName
    res ← runForm (forumTopicForm memberName email) $ Right noHtml
    case res of
        Left xhtml → simplePage "New Forum Topic" forumDeps
            [thediv ![theclass "comments"] << xhtml]
        Right (title, (body, _)) → do
            r ← createTopic (title, body) n
            case r of
                Nothing → simplePage "Error Creating Forum Topic" forumDeps []
                Just _ → redirect $ "../" ++ n

```

The form for creating the topic is very simple. All we need is a title and the body of the first comment (topics can't be created without a root comment).

```

forumTopicForm :: String → Maybe String →
    AppForm (String, (String, Maybe Integer))
forumTopicForm memberName email =
    commentBox ((,) < $ > plug (thediv <<) ("Topic Title" 'formLabel'
        (plug (λxhtml → thediv ![theclass "title"] << xhtml) $
            F.input Nothing) 'check' ensures
            [((>0) ∘ length, "Title must not be empty."),
             ((≤ 80) ∘ length, "Title must be 80 characters or shorter.")])
        < * > commentForm memberName email Nothing

```

This creates the topic in the database given the title and root comment body.

```

createTopic :: (String, String) → String → App (Maybe Integer)
createTopic (t, c) fn = do
    memberNo' ← asks appMemberNo
    case memberNo' of
        Nothing → return Nothing
        Just memberNo → withTransaction' $ do
            c' ← asks appDB
            commentNo ← storeComment memberNo c Nothing
            case commentNo of
                Nothing → liftIO $ rollback c' >> throwDyn ()
                Just n → do
                    r ← quickStmnt' "INSERT INTO forum_topic "

```

```

"(forum_name, title, root_comment, last_comment) "
"VALUES (?, ?, ?, ?)"
      [toSql fn, toSql t, toSql n, toSql n]
  case r of
    Nothing → liftIO $ rollback c' >> throwDyn ()
    Just _  → (liftIO memcacheFlush) >> return n

```

The following HTML is pretty messy. I now know why threaded comments are rare: they're difficult to implement! It's not just the database threading that's difficult but the display as well.

```

commentBox :: Monad m => XHtmlForm m a → XHtmlForm m a
commentBox = plug (\xhtml → thediv! [theclass "comment toplevel editable"] <<
  xhtml)

```

Creating the form for a comment requires knowing the comment's parent (as a comment number). If this comment is not a reply to another comment (in the case of root comments), it could be passed as Nothing.

The comment form displays the member's gravatar as a visual hint to what the comment will eventually look like when posted.

```

commentForm :: String → Maybe String → Maybe String →
  AppForm (String, Maybe Integer)
commentForm _ email parent = plug (\xhtml → concatHtml [
  image! [width "60", height "60", theclass "avatar",
    src $ gravatarWith (maybe "" (map toLower) email)
      Nothing (size 60) (Just "wavatar")],
  thediv! [theclass "speech soft"] << xhtml,
  thediv! [theclass "signature"] << [
    helpButton "http://daringfireball.net/projects/markdown/basics"
      (Just "Formatting Help"),
    isJust parent ? button << "Preview" + + + stringToHtml " " + + +
      submit "" "Send Reply"
      $ submit "" "Create"]])
  ((λa b → (a, maybeRead <=< b)) < $ > (F.textarea Nothing 'check' ensures
    [ ((>0)      o length, "Comment must not be empty."),
      ((≤ 10000) o length, "Comment must be 10,000 characters "
        "or shorter.")])
    < * > (nothingIfNull $ F.hidden parent))

```

We're finally getting to the core of the forum: individual topic pages. The forum topic page displays the entire comment thread. Eventually it will probably need paging.

```

forumTopicPage :: String → Integer → App CGIResult
forumTopicPage fn i = do
  r ← queryTuple' "SELECT t.root_comment, t.title, f.title "
"FROM forum_topic t, forum f "
"WHERE f.name = t.forum_name "
"AND t.topic_no = ?" [toSql i]
  case r of
    Just [root, title, fTitle] → do

```

```

    comments ← queryTuples'
    "SELECT c.comment_no, t.level, m.username, m.email, "
"c.time, c.comment "
"FROM comment c, member m, "
"connectby('comment', 'comment_no', 'parent_no', ?, 0) "
"AS t(comment_no int, parent_no int, level int) "
"WHERE c.comment_no = t.comment_no "
"AND m.member_no = c.author" [root]
    case comments of
    Nothing → error "Error retrieving comments."
    Just cs → do
        commentsHtml ← mapM (displayForumComment i) cs
        stdPage (fromSql title) (forumDeps ++ [JS "forum-comment"]) [] [
            breadcrumbs [
                anchor! [href "../forums"] << "Forums",
                anchor! [href $ "../" ++ fn] << (fromSql fTitle :: String),
                stringToHtml $ fromSql title],
            thediv! [theclass "comments"] << commentsHtml]
    _ → output404 ["forum", fn, show i]

```

Displaying forum comments is complicated by the fact that we insert hidden comment forms along with each comment if the page is going to a confirmed member. This allows instant dynamic interactivity. However, it comes at the cost of inflated HTML pages.

There is a more efficient way to do this that involves more JavaScript, but for now we keep it simple.

```

displayForumComment :: Integer → [SqlValue] → App Html
displayForumComment i [n, l, u, e, t, c] = do
    memberName ← asks appMemberName
    email ← asks appMemberEmail
    let n' :: Integer    = fromSql n
        l' :: Integer    = fromSql l
        u' :: String     = fromSql u
        e' :: String     = e ≡ SqlNull ? "" $ fromSql e
        t' :: UTCTime    = fromSql t
        c' :: String     = fromSql c
        id'              = "reply-" ++ (show n')
    reply ← case (memberName, email) of
        (Just mn, Just _) → do
            (_, xhtml) ← runForm' $ commentForm mn email (Just $ show n')
            return $ concatHtml [
                button! [theclass $ "reveal " ++ id'] << "Reply",
                paragraph! [thestyle "clear: both"] << noHtml,
                thediv! [thestyle "display: none",
                    identifier id',
                    theclass "reply"] << [
                        thediv! [theclass "comment editable"] <<
                            form! [method "POST"] << [hidden "topic" (show i),
                                xhtml]]]
        (Just _, Nothing) → return $ anchor! [href "/member/confirmation"] <<

```

```

        "Confirm Your Email to Reply"
    -   → return $ anchor ! [href "/member/login"] <<
        "Login to Reply"
return $ thediv ! [theClass "comment toplevel",
                 thestyle $ "margin-left:" ++ (show $ l' * 2) ++ "em"] << [
    paragraph ! [theClass "timestamp"] << formatSimpleTime t',
    image ! [width "60", height "60", theClass "avatar",
            src $   gravatarWith (map toLower e')
                Nothing (size 60) (Just "wavatar")],
    thediv ! [theClass "speech"] << displayCommentBody c',
    thediv ! [theClass "signature"] << ("- " ++ u'),
    thediv ! [theClass "reply-options"] << reply]
displayForumComment _ _ = return $ paragraph << "Error retrieving comment."

```

Each comment uses (Pandoc-extended) Markdown syntax.

```

displayCommentBody :: String → Html
displayCommentBody = markdownToHtml

```

Storing a comment establishes and returns its unique comment number.

```

storeComment :: Integer → String → Maybe Integer → App (Maybe Integer)
storeComment memberNo body parent = do
    c ← asks appDB
    liftIO $ insertNo c "INSERT INTO comment (author, comment, parent_no) "
    "VALUES (?, ?, ?)" [toSql memberNo, toSql body, toSql parent]
    "comment_comment_no_seq"

```

Replying to a comment is also a complex matter (are you noticing a trend here?). The complexity is mainly because we need to update information in a couple relations. Determining the number of comments or the time of the latest comment in a thread is possible with SQL, but it can be expensive. So when adding a comment to a thread we update the number of comments in the thread and the last comment time.

If the comment is posted successfully, we need to remove the topic page from the cache so that it gets regenerated on the next request.

```

replyToForumComment :: App CGIResult
replyToForumComment = do
    memberName ← asks appMemberName
    email ← asks appMemberEmail
    topicNum ← fromJust ◦ maybeRead < $ > getRequiredInput "topic"
    case (memberName, email) of
        (Just mn, Just _) → do
            parent ← getInput "parent"
            res ← runForm (commentForm mn email parent) $ Right noHtml
            case res of
                Left xhtml → outputJSON [("html", showHtmlFragment $ thediv !
                    [theClass "comment editable"] << xhtml),
                    ("status", "incomplete")]
                Right (body, parent') → do
                    memberNo ← fromJust < $ > asks appMemberNo

```

```

    res' ← withTransaction' $ do
      commentNo ← storeComment memberNo body parent'
      run' "UPDATE forum_topic "
"SET last_comment = ?, "
"num_replies = num_replies + 1 "
"WHERE topic_no = ?"
      [toSql commentNo, toSql topicNum]
      res'' ← queryTuple' "SELECT c.comment_no, 0 as level, "
"m.username, m.email, "
"c.time, c.comment "
"FROM comment c, member m "
"WHERE m.member_no = c.author "
"AND comment_no = ?"
      [toSql commentNo]
      liftIO ◦ commit ≪ asks appDB
      return $ fromJust res''
case res' of
  Nothing → outputJSON [("html", "Error posting comment."),
                        ("status", "error")]
  Just c → do
    c' ← asks appDB
    liftIO $ commit c'
    liftIO memcacheFlush
    comment ← displayForumComment topicNum c
    outputJSON [("html", showHtmlFragment comment),
               ("status", "accepted")]
    → outputUnauthorized

```

Previewing comments is a truly asynchronous process (the first one implemented). It makes for complicated JavaScript but a smoother interface.

We need to make sure that this doesn't go out of sync with displayComment.

Also, does this lead to XSS vulnerabilities?

```

commentPreview :: App CGIResult
commentPreview = do
  memberName ← asks appMemberName
  case memberName of
    Nothing → outputUnauthorized
    Just _ → do
      comment ← getRequiredInput "comment"
      outputJSON [("html", showHtmlFragment $ displayCommentBody comment),
                 ("status", "OK")]

```

That's it! You've seen everything required to run [http://www.vocabulink.com/!](http://www.vocabulink.com/)