TYPE-SAFE WEB DEVELOPMENT

with Yesod and Haskell

Goals of this talk

- Can't cover everything about yesod in one talk.
- To give you a taste of type safe web dev.
- To peek under the hood and get an understanding on how the type-safe parts work.
- This taste test should help you figure out whether:
  - You love it and want to learn more.
  - You want to run away screaming (!!).

Talk overview

- Brief introduction to web dev & the yesod project
- Introduction to generated handler / route code.
- Break for pizza
- Exploration of a bigger app (with forms and DB)
- Demo of type-safety when refactoring our bigger app.
- Conclusions and Questions

Feel free to jump in and ask questions during the talk! I'd prefer to have a bit of a conversation and have people learn rather than just having me drone on! ;)

Yesod: An Introduction

- Yesod means ‘foundation’ in Hebrew
- Written in Haskell ☺
- Created initially by Michael Snoyman
- Has grown a long way over the past 3 years.
- Recently hit 1.0 major version.

Yesod: core tenets

- Type-safety
- Conciseness
- Performance
- Modularity

Webdev in a nutshell: input

- Resource identified / located by URI.
  - http://example.com/examples/1
- Method
- Query Parameters
- RequestBody + Content-Type
- Client accepted output types
- Cookies
- Etc.

Webdev in a nutshell: do stuff!

- Make a decision to do something based on input
  - Load a static file from disk
  - Load / Update / Delete something from a database.
  - Fire the missiles
  - (most of the things that we want to do will be in IO)

Webdev in a nutshell: output

- Status Code
  - 2xx yay, it worked!
  - 3xx the princess is in another castle
  - 4xx client error
  - 5xx server error
- Response Body + Content Type
  - Static file resource
  - Use data to dynamically generate HTML / JSON / XML.

Boundary Problem

- On each side of our code (HTTP requests, DB) we have messy, un-typed strings.
- We would like static types in our programs.
- We'd love a framework that automatically:
  - Converts the external strings into our types.
  - Converts our types back into strings in a predictably safe manner.

Typographical Conventions

- Code that is written by us:

```
getHomeR = defaultLayout $ [whamlet|Hello Yesod!|]
```

- Generated Code (italics for contextual handwritten code):

```
getHomeR = defaultLayout $
  toWidget $ (preEscapedText . pack) "Hello yesod!"
```

Core Yesod

Introducing the core of every yesod app:

- Configuration: Foundation Datatype & Yesod type class instance
- Handlers
- Generated code from our routes definition.

Hello Yesod!

```
{-# LANGUAGE TypeFamilies, QuasiQuotes, MultiParamTypeClasses,
             TemplateHaskell, OverloadedStrings #-}
import Yesod
import Data.Text

data HelloWorld = HelloWorld

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

instance Yesod HelloWorld

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

main = warpDebug 3000 HelloWorld
```

Language Pragmas

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

- These extend Haskell98 to provide syntactic features.
- Needed to generate code and to make the generated code compile.
- In a real yesod project you set these in your .cabal file rather than in every file.

Foundation Type

```
data HelloWorld = HelloWorld
```

- Just a normal Haskell data type. Can be accessed from anywhere inside your handlers using getYesod.
- Used for storing data that is central to your app:
  - Configuration
  - Connection pools
  - IORefs (!)
  - Whatever you want
- Touch a bit more on this later.

Yesod Instance

```
instance Yesod HelloWorld
```

- Can override methods in this instance to configure various things about yesod for the HelloWorld foundation type. Using defaults here.
- Not shown here, but there are many other type classes used for configuration. More on this in our bigger example later.

Handlers

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

- In MVC speak, this is our controller.
- Handler is a monad that encapsulates an entire HTTP request/response.
- Is a transformer built up of a few monads, IO being the most important. Can lift any IO into handler.
- The RepHtml part of the type signature lets yesod know that this handler returns a HTML response only.

Widgets

```
getHomeR = defaultLayout [whamlet|Hello yesod!|]
```

- In MVC speak, widgets are our View.
- Widget is just a monad that sequences and collects CSS, JS & HTML.
- Templating languages that create widgets:
  - Julius (Javascript)
  - Cassius (CSS)
  - Hamlet (HTML)
- Widgets can compose together (awesome!)

Routes

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

- At the base URI of our app create a route HomeR.
- Accept the GET method only.
- Calls to “GET /” will be dispatched to getHomeR handler.
- HomeR suffix is just a convention. R stands for Route.
- More on this later.

Running your app

```
main = warpDebug 3000 HelloWorld
```

- Starts up a web server that listens for connections on port 3000.
- Since this is in a main, we can run our app like this:
  - `runhaskell helloworld.0.hs`

- (non-Debug) Warp is the preferred deployment solution.
- Some don't even bother putting a frontend in front of it.
- Can deploy to fast CGI instead if you don't trust warp.
- Warp is not part of yesod. Is a separate webserver in its own right.

- Lets test it: http://localhost:3000/

mkYesod & parseRoutes

Jumping into the generated code to see how we get most of the Yesod type-safety for free.

mkYesod & parseRoutes

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

- parseRoutes is a QQ converts text to a route AST. There is also parseRoutesFile for reading from a txt file.
- mkYesod is where the magic happens.

Handler / Widget Type Aliases

```
type Handler = GHandler HelloWorld HelloWorld
type Widget = GWidget HelloWorld HelloWorld ()
```

- Generates handler & widget type aliases for your foundation data type.
- Better than writing the full types everywhere.
- Yesod has concept of django-like subsites.
- For an admin subsite: GHandler HelloWorld Admin

RenderRoute HelloWorld

```
instance RenderRoute HelloWorld where
  data Route HelloWorld = HomeR deriving (Show, Eq, Read)
  renderRoute HomeR = ([], [])
```

- TypeFamilies GHC extension allows data declaration inside type class.
- Creates:
  - Constructors for our routes.
  - Function to change a route into a URL
- More on why this is important later!

Yesod Dispatch: All of this code for one route!

- Don't read this. Just be glad that you don't have to type it!

```
instance YesodDispatch HelloWorld HelloWorld where
  yesodDispatch master sub toMaster
    handler404 handler405 method pieces =
    case dispatch pieces of
      Just f  -> f master sub toMaster handler404 handler405 method
      Nothing -> handler404
    where
      dispatch = Yesod.Routes.Dispatch.toDispatch
                 [ Yesod.Routes.Dispatch.Route [] False handleHomePieces ]
      handleHomePieces [] = Just handleHomeMethods
      homeHomePieces _ = error "Invariant violated"
      handleHomeMethods master sub toMaster app handler405 method =
        case Data.Map.lookup method methodsHomeR of
          Just handler -> yesodRunner handler master sub (Just HomeR) toMaster
          Nothing -> handler405 HomeR
      methodsHomeR = Data.Map.fromList
                     [ ( ( pack "GET" ) , ( fmap chooseRep getHomeR ) ) ]
```

Yesod Dispatch

```
instance YesodDispatch HelloWorld HelloWorld where
  yesodDispatch master sub toMaster
    handler404 handler405 method pieces =
```

- Takes the method and URI pieces and does this:
  - If the pieces aren't [], return 404.
  - If the pieces are [] and the method isn't GET, 405.
  - If the pieces are [], and the method is get: route to the getHomeR handler.

Parameterised Routes

This is where our type safe routes start to get interesting.

Parameterized greetings

```
mkYesod "HelloWorld" [parseRoutes|
/hello/#Text HomeR GET
|]
getHomeR :: Text -> Handler RepHtml
getHomeR name = defaultLayout [whamlet|Hello #{name}!|]
```

- Routes can have dynamic pieces.
- Here we have given the piece type Text (Data.Text)
- That dynamic piece is passed into the handler function.
- It is a compile error if your handler is missing that parameter or it is the wrong type.
- #{ … } interpolates a string into the HTML (but HTML escapes before doing so).
- Lets test this: http://localhost:3001/hello/bfpg

Generated RenderRoute

```
instance RenderRoute HelloWorld where
  data Route HelloWorld = HomeR Text deriving (Show, Eq, Read)
  renderRoute (HomeR name) =
    ([pack "hello",toPathPiece name], [])
```

- Our HomeR constructor now has a Text argument.
- renderRoute has a funny toPathPiece in it.

Generated YesodDispatch

```
handleHelloPieces [_, x ] = do
  y <- fromPathPiece x
  Just $ handleHelloMethods y
```

- A bit of the dispatch code that will return Nothing if x can't be converted into the desired type.
- A return of nothing will cause a 404.
- This snippet doesn't care about the first piece since it wont get to here unless the first piece was "hello".

Path Pieces

- :i PathPiece

```
class PathPiece s where
  fromPathPiece :: Text -> Maybe s
  toPathPiece :: s -> Text
      -- Defined in `Web.PathPieces'
instance PathPiece String -- Defined in `Web.PathPieces'
instance PathPiece Text -- Defined in `Web.PathPieces'
instance PathPiece Integer -- Defined in `Web.PathPieces'
instance PathPiece Int -- Defined in `Web.PathPieces’
```

- Anything that has a PathPiece instance can be a validated , type-safe part of a URL.
- Ad hoc polymorphism allows us to create PathPieces of any type (Dates, Enums , Etc).

Pizza Break!!

Does anyone have any questions while we wait for pizza?

Complete App

Introduction to our ‘larger scale’ app and the Yesod code that we need to write to make it happen.

But first, a demo!

- An app to create and show notes.
- Lets see it in action: http://localhost:3002/notes
- Now we're going to go through the interesting bits of code that make this happen.

Using our foundation data type!

```
data Notes = Notes {
  dbConn :: Connection -- Use a connection pool in prod, plz.
}
instance YesodPersist Notes where
  type YesodPersistBackend Notes = SqlPersist
  runDB f = do
    conn <- fmap dbConn getYesod
    runSqlConn f conn
```

- runDB happens inside a handler, so can "getYesod"
- runDB is much handier than extracting the connection every time.

Our Model / Schema

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persist|
Note
  title Text
  date  Day Maybe
  body  Textarea
|]
```

- Pretty simple and declarative due to metaprogramming.
- Creates a Note datatype with record accessors using tabel and column name (e.g. noteDate grabs the date out of a Note).
- Automatically generates a NoteId column. Field is of type NoteId
- Maybe marker makes date nullable in the DB and the data type a Data.Maybe Day instead of just Day.

Note Routes

```
mkYesod "Notes" [parseRoutes|
/notes/          NotesR GET POST
/notes/#NoteId NoteR  GET
|]
```

- Our autogenerated NoteId has a PathPiece instance generated by the “share” TH function.
- Since we can only look for Notes with a NoteId, we can’t accidentally load a User with the id that gets passed into our hander.

Note Creation Form

```
createNoteForm today = renderDivs $ Note
  <$> areq textField "Title" Nothing
  <*> aopt dueDateField "Date" ( Just ( Just today ) )
  <*> areq textareaField "Body" Nothing
  where
    dueDateField = jqueryDayField def
```

- A form that has all the information it needs to:
  - Generate a widget with the form HTML, JS and CSS inside.
  - Process the form inputs that were posted back and give us a validation error or a fully constructed note.
- Note that the form fields are strongly typed. It would be a type error:
  - To specify the dueDateField for the Title or Body fields.
  - To give something other than a Day as the default value for the date field.
- Types are really giving us a big win here.

GET /notes

```
getNotesR = do
  today <- fmap utctDay $ liftIO getCurrentTime
  (widget, encType) <- generateFormPost $ createNoteForm today
  showCreateNoteForm widget encType
```

- Get the current time in IO
- Generate the widget for the form
- Pass it to showCreateForm

Displaying our form (and list)

```
showCreateNoteForm widget encType = do
  notes  <- runDB $ selectList [] [Asc NoteTitle]
  defaultLayout [whamlet|
    <h1>Notes
    <ul>
      $forall Entity id note <- notes
        <li><a href="@{NoteR id}">#{noteTitle note}</a>
    <h1>Create Note
    <form method=post action=@{NotesR} enctype=#{encType}>
      ^{widget}
      <input type=submit>
```

- Note optional closing tags. Format adheres to haskell offside indentation rule.
- @{ … } will make a URL to any Route. Note that we had to supply the id to make a NoteR
- ^{ … } composes a widget into this one, including merging CSS and JS.
- The generated forms are designed to be composed, so they don't add a form element or submit button.

Processing the form POST

```
postNotesR :: Handler RepHtml
postNotesR = do
  today <- fmap utctDay $ liftIO getCurrentTime
  ((result, widget), encType) <-
    runFormPost $ createNoteForm today
  case result of
    FormSuccess note -> postNotesRWin note
    _ -> showCreateNoteForm widget encType

postNotesRWin note = do
  noteId <- runDB $ insert note
  redirect $ NoteR noteId
```

- FormSuccess yields a constructed note, which is easily inserted. We then redirect to the note.
- Failed form means that there are error messages against the bad fields in the widget.

Showing a Note

```
getNoteR id = do
  note <- runDB $ get id
  case note of
    Nothing     -> notFound
    Just ( Note title date body ) -> do
      defaultLayout [whamlet|
        <h1>#{ title }
        $maybe d <- date
          \ ( dated: #{ show d } )
        <p>#{ body }
      |]
```

- Returning a 404 if the note doesn't exist is easy!
- Note that we have to convert the day into a string.

Last little-but-important bits.

```
main = withSqliteConn ":memory:" run
  where
    run conn = do
      runSqlConn (runMigration migrateAll) conn
      warpDebug 3002 (Notes conn )
```

- Create an in memory sqlite DB
- Get persistent to setup the schema for us on startup

Generated Code for notes.hs

An exploration of the expanded whamlet QQ so that we can get an idea of the type safety with regard to safely escaped text.

Whamlet QQ Expansion

```
showCreateNoteForm widget encType = do
  notes  <- runDB $ selectList [] [Asc NoteTitle]
  defaultLayout $ do
    toWidget $ ( preEscapedText . pack) "<h1>Notes</h1><ul>"
    Data.Foldable.mapM_
      (\ (Entity id note ) -> do
          toWidget $ (preEscapedText . pack) "<li><a href=\""
          ((lift getUrlRenderParams) >>= (\ urender ->
            toWidget (toHtml (urender (NoteR id) [] ))))
          toWidget $ (preEscapedText . pack) "\">"
          toWidget (toHtml (noteTitle note))
          toWidget $ (preEscapedText . pack) "</a></li>" )
      notes
```

- This is just the part that printed our list of note links!

Expanded Whamlet QQ

- Code is pretty messy. Glad that we don't have to write it.
- What is it giving us? Lets look into the different patterns of generated code to see.
- Four main patterns in the code generation:
  - Literal HTML in hamlet
  - #{ … } string interpolation
  - @{ … } route interpolation
  - ^{ … } widget interpolation

Whamlet Literal HTML

```
[whamlet|
    <h1>Notes
      <ul>
```

```
toWidget $ ( preEscapedText . pack) "<h1>Notes</h1><ul>"
```

- Using Text.Blaze preEscaped test. Literal HTML is appended to widget.

Whamlet String Interpolation

```
#{noteTitle note}
```

```
toWidget (toHtml (noteTitle note))
```

- Using Text.Blaze toHtml. Any string interpolation is automatically escaped.
- There are ways to put haskell strings un-escaped into a widget. They are just rightfully not in plain sight.

Whamlet Route Interpolation

```
@{NoteR id}
```

```
((lift getUrlRenderParams)
  >>=
  (\ urender -> toWidget (toHtml (urender (NoteR id) [] ))))
```

- Using Text.Blaze preEscaped test. Literal HTML is appended to widget.
- Note since you can construct routes with un-escaped strings, yesod escapes rendered URI.

Whamlet Widget Interpolation

```
^{widget}
```

```
toWidget widget
```

- Just appends this widget to our widget.
- toWidget (method of the Widget monad) does all the magic of appending the CSS,JS & HTML together.

Templating Languages

- Give you automated ways of keeping your interpolated strings escaped and safe.
- Very minimal boilerplate due to QQ.
- Templating languages add some neat features to target language.
- Have the freedom to put in un-escaped text if that is what you really need to do.

Type-Safety Demo

Lets jump into the note app and break stuff!

Add an Year/Month to Note Route

```
maybeRead :: Read a => String -> Maybe a
maybeRead =  fmap fst . listToMaybe . Reads

newtype Year  = Year Int  deriving (Show,Read,Eq)
newtype Month = Month Int deriving (Show,Read,Eq)

instance PathPiece Year where
  toPathPiece   = pack . show
  fromPathPiece = maybeRead . unpack

instance PathPiece Month where
  toPathPiece   = pack. show
  fromPathPiece = (F.find validMonth) . maybeRead . unpack
    where validMonth (Month m) = ( m >= 1 && m <= 12 )

mkYesod "Notes" [parseRoutes|
/notes/          NotesR GET POST
/notes/#Year/#Month/#NoteId NoteR  GET
|]
```

Types to the Rescue! Handler Update

```
getNoteR id = do
  -- <snip>
```

```
notes.broken.1.hs:48:1:
    The function `getNoteR' is applied to three arguments,
    but its type `NoteId -> Handler RepHtml' has only one
    <snip>
```

Types to the Rescue! Note Links

```
showCreateNoteForm widget encType = do
  notes  <- runDB $ selectList [] [Asc NoteTitle]
  defaultLayout [whamlet|
    <h1>Notes
    <ul>
      $forall Entity id note <- notes
        <li><a href="@{NoteR id}">#{noteTitle note}</a>
```

```
notes.broken.1.hs:75:26:
    Couldn't match expected type `Route master0'
                with actual type `Month -> NoteId -> Route Notes'
    In the return type of a call of `NoteR'
    Probable cause: `NoteR' is applied to too few arguments
    <snip>
```

Types to the rescue! Redirect Broken

```
postNotesRWin note = do
  noteId <- runDB $ insert note
  redirect $ NoteR noteId
```

```
notes.broken.1.hs:95:21:
    Couldn't match type `Key (YesodPersistBackend master0) val0'
                        with `Year'
      In the return type of a call of `insert'
      In the second argument of `($)', namely `insert note'
      In a stmt of a 'do' block: noteId <- runDB $ insert note
```

- Type inference getting tripped up and making our missing route params more evil looking than it is.

Beyond these examples

A quick mention of neat features that we couldn't cover in the talk.

Subsites & Static Subsite

- Subsites feel a bit like Django.
- There is a predone Auth subsite which has a lot of bells and whistles.
- Static subsite generates routes for everyone of your static assets.
  - Really cool because you can use routes to static images (in JS,CSS & HTML) and have compile errors if you rename / move them.
  - Can host the static files on you app server or get the subsite to generate URLs to your CDN.

Composable Handlers / Widgets

- Had a brief peak at this, but worth mentioning again.
- Because widgets compose their styles and JS together, it allows you to write very modular views.

Auth, Sessions and all the boring bits

- Has auth right out of the box.
  - Google ID
  - Facebook ID
  - BrowserID
  - User/Pass DB auth.
- Also has client side sessions. No server side sessions (this is a good thing).

Questions

Open Questions to Discuss

- Type errors are sometimes tricky to figure out. Forces you to know about the innards of generated code, sometimes. Is this a necessary evil, or worth it?
- All of your handlers – thus most of you code – are actually the IO monad in disguise. Didn't us Fpers already learn that trying to compose things that do IO is hard and error prone?!?

Further Information

- Sample code and slides:
  - https://github.com/benkolera/bfpg-yesod/
- Yesod Website: www.yesodweb.com
  - Great screen cast
  - Online version of the book
- Book in print and ebook: http://shop.oreilly.com/product/0636920023142.do
- Haskellers website: http://www.haskellers.com/