Building a Blogging System

with Haskell and Yesod

Gavin Whelan

May 23, 2017

About Me

IU Computer Science Graduate
LambdaConf Hunger Prevention Manager

My site: ambientmemory.com
GitHub: @gavwhela

CTO Whiteboard Dynamics LLC
Company site: whiteboarddynamics.co

Attributions

This presentation uses resources from the Yesod Book,
https://github.com/yesodweb/yesodweb.com-content
http://www.yesodweb.com/book

as well as the yesod-postgres template.
MIT License

Table of Contents

1. Basic Background
   - Why Yesod?
   - Language Extensions
   - Hello World
2. Set up Codio & Dive into an exercise
3. Content Creation
4. Handlers that actually do something
5. Industrial Structure
6. Authentication and Authorization
7. Forms
8. Persistence

Why Yesod?

Why Yesod?

- Plays to the strengths of Haskell
  - Well defined types to prevent unsafe data use
  - Referential transparency preventing unintended side effects

Why Yesod?

- Plays to the strengths of Haskell
  - Well defined types to prevent unsafe data use
  - Referential transparency preventing unintended side effects
- Saves typing
  - Forms leverage Applicatives to reduce boilerplate
  - Terse DSLs for route definitions and database entities
  - Code generation to write correct and consistent code for you

Why Yesod?

- Plays to the strengths of Haskell
  - Well defined types to prevent unsafe data use
  - Referential transparency preventing unintended side effects
- Saves typing
  - Forms leverage Applicatives to reduce boilerplate
  - Terse DSLs for route definitions and database entities
  - Code generation to write correct and consistent code for you
- Performant
  - Templating approaches allow HTML, CSS, and Javascript to be optimized at compile time, by GHC
  - Advanced underlying technologies such as conduits and builders allow code to run in constant memory
  - Flagship web server Warp is highly scalable, using lightweight user threads running on multiple processes, receiving IO events with epoll (or kqueue)

Why Yesod?

- Plays to the strengths of Haskell
  - Well defined types to prevent unsafe data use
  - Referential transparency preventing unintended side effects
- Saves typing
  - Forms leverage Applicatives to reduce boilerplate
  - Terse DSLs for route definitions and database entities
  - Code generation to write correct and consistent code for you
- Performant
  - Templating approaches allow HTML, CSS, and Javascript to be optimized at compile time, by GHC
  - Advanced underlying technologies such as conduits and builders allow code to run in constant memory
  - Flagship web server Warp is highly scalable, using lightweight user threads running on multiple processes, receiving IO events with epoll (or kqueue)
- Modular
  - Many general purpose packages have spun off of Yesod and can be used independently

Language Extensions

Overloaded Strings

```haskell
class IsString a where
    fromString :: String -> a
```

Overloaded Strings

```
class IsString a where
    fromString :: String -> a
```

Without OverloadedStrings

```
"Literal String" :: String
"Literal String" :: [Char]
```

Overloaded Strings

```haskell
class IsString a where
    fromString :: String -> a
```

Without OverloadedStrings

```haskell
"Literal String" :: String
"Literal String" :: [Char]
```

With OverloadedStrings

```haskell
{-# LANGUAGE OverloadedStrings #-}
"Literal String" :: IsString a => a
```

Type Families

```haskell
{-# LANGUAGE TypeFamilies #-}
import Data.Word (Word8)
import qualified Data.ByteString as S

class SafeHead a where
    type Content a
    safeHead :: a -> Maybe (Content a)

instance SafeHead [a] where
    type Content [a] = a
    safeHead [] = Nothing
    safeHead (x:_) = Just x

instance SafeHead S.ByteString where
    type Content S.ByteString = Word8
    safeHead bs
        | S.null bs = Nothing
        | otherwise = Just $ S.head bs
```

Template Haskell

```haskell
{-# LANGUAGE TemplateHaskell #-}

-- | Template Haskell (TH) is Haskell's approach to code
-- generation. Yesod uses TH to reduce boilerplate. Code from
-- before a TH splice cannot refer to code within the TH, or what
-- follows.

$(templateHaskellFunction arg1 arg2)

-- | Or at the top level
templateHaskellFunction arg1 arg2
```

QuasiQuotes

```haskell
{-# LANGUAGE QuasiQuotes #-}

-- | QuasiQuotes (QQ) allows arbitrary content to be taken inline
-- in Haskell code and processed, generating Haskell code as TH does.

-- | The name of the quasi-quoter is given between the opening
-- bracket and the first pipe. The quasi-quoted region is closed
-- with |]

[hamlet|<p>This is quasi-quoted Hamlet.|]
```

Hello World

Hello World

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes       #-}
{-# LANGUAGE TemplateHaskell   #-}
{-# LANGUAGE TypeFamilies      #-}
import Yesod

data HelloWorld = HelloWorld

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

instance Yesod HelloWorld

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

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

Hello World: Result

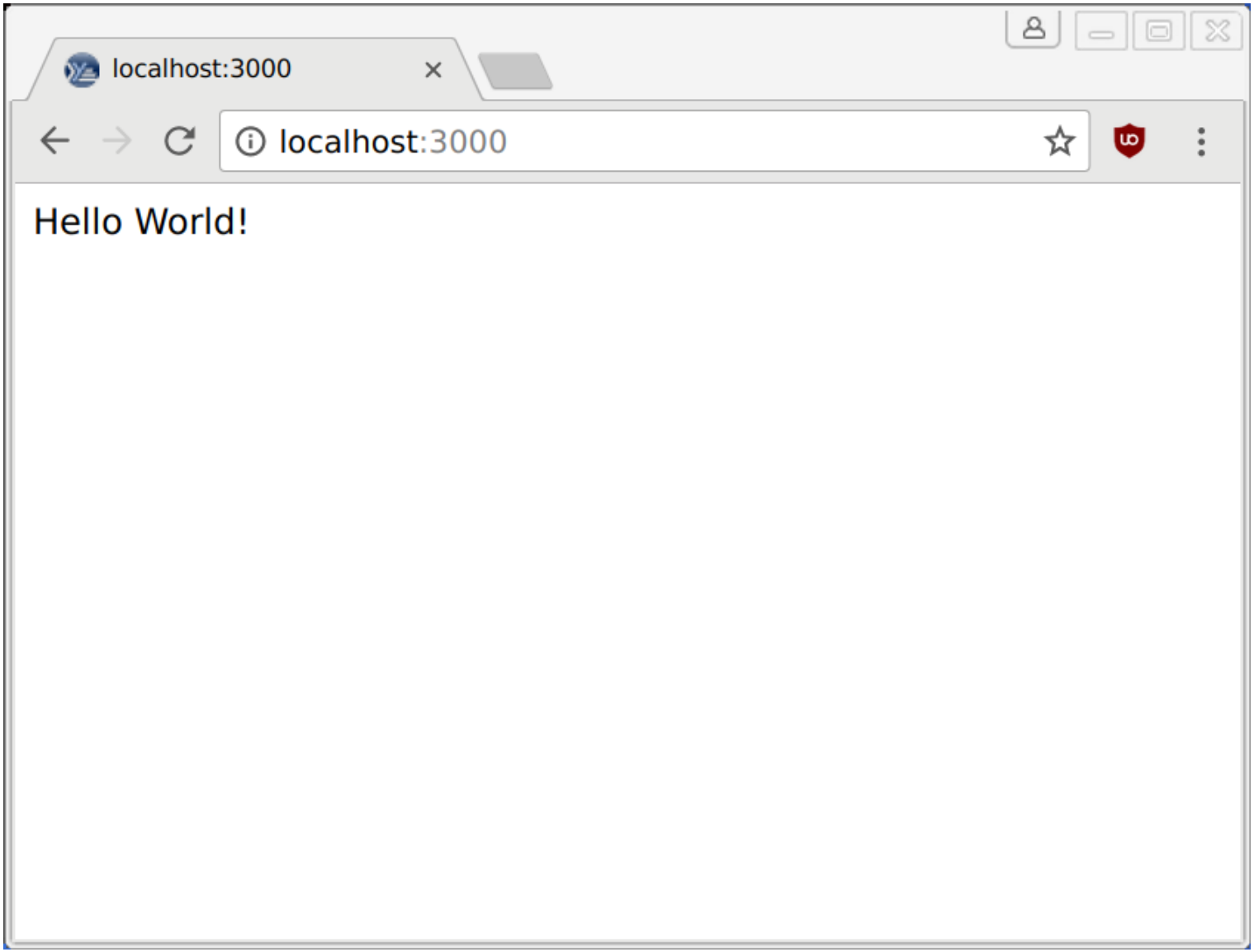

Table of Contents

1. Basic Background
2. Set up Codio & Dive into an exercise
   - Codio
   - First Exercise
   - What did we do?
3. Content Creation
4. Handlers that actually do something
5. Industrial Structure
6. Authentication and Authorization
7. Forms
8. Persistence

Codio

Setting up Codio

Sign in to Codio

Join the class

Enter the first unit

Should see readme

First Exercise

First Exercise

```
{-# LANGUAGE OverloadedStrings, QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell, TypeFamilies  #-}
import Yesod

data Links = Links

mkYesod "Links" [parseRoutes|
/       HomeR  GET
/page1 Page1R GET
/page2 Page2R GET
|]

instance Yesod Links

getHomeR  = defaultLayout [whamlet|<a href=@{Page1R}>Go to page 1!|]
getPage1R = defaultLayout [whamlet|<a href=@{Page2R}>Go to page 2!|]
getPage2R = defaultLayout [whamlet|<a href=@{HomeR}>Go home!|]

main = warp 3000 Links
```

What did we do?

Congratulations!

Congratulations, you've just added a new page to a Yesod application!

So far you've already used:

- Monads (hopefully you weren't scared of those)
- Shakespearean Templating
- Routing DSL
- Typesafe URLs
- Handlers
- Quasi-Quoters
- And more!

Table of Contents

1. Basic Background
2. Set up Codio & Dive into an exercise
3. Content Creation
   - Shakespearean Templating
   - Widgets
   - Exercise Two
4. Handlers that actually do something
5. Industrial Structure
6. Authentication and Authorization
7. Forms
8. Persistence

Shakespearean Templating

Shakespearean Templating

General purpose templating languages

Hamlet	(HTML)
Lucius	(CSS)
Cassius	(CSS)
Julius	(JavaScript)

Interpolation

```
#{myVar}
```

Variable interpolation

Interpolation

`#{myVar}`	Variable interpolation
`@{HomeR}`	URL interpolation

Interpolation

```
#{myVar}
```
Variable interpolation

```
@{HomeR}
```
URL interpolation

```
@?{(HomeR, [("token", validateToken)]}
```
↑ With query parameters

Interpolation

```
#{myVar}
@{HomeR}
@?{(HomeR, [("token", validateToken)]}
^{footer}
```

Variable interpolation
URL interpolation
↑ With query parameters
Template Embedding

Hamlet

- Whitespace based, no closing tags

Hamlet

- Whitespace based, no closing tags
- Interpolations

Hamlet

- Whitespace based, no closing tags
- Interpolations
- Quality of life improvements

Hamlet

- Whitespace based, no closing tags
- Interpolations
- Quality of life improvements
- Embedded logic, looping, and pattern matching

Hamlet

- Whitespace based, no closing tags
- Interpolations
- Quality of life improvements
- Embedded logic, looping, and pattern matching
- Optional attributes

Hamlet

```
$doctype 5
<html>
    <head>
        <title> <a href=@{HomeR}> #{pageTitle} - My Site </a>
        <link rel=stylesheet href=@{Stylesheet}>
    <body #body>
        <h1 .page-title>#{pageTitle}
        <p>Here is a list of your friends:
        $if null friends
            <p>Sorry, I lied, you don't have any friends.
        $else
            <ul>
                $forall Friend name age <- friends
                    <li>#{name} (#{age} years old)
```

Template interpolation

```
wrapper :: HtmlUrl a -> HtmlUrl a
wrapper content =
  [hamlet|
      <html>
          <head>
              <title> My Site
          <body>
            ^{content}
  |]

content :: HtmlUrl a
content =
  [hamlet|
    <h1> Welcome
    <p> This is the content hamlet!
  |]
```

More Embedded Logic

Attributes:

```
<input type=checkbox :isChecked:checked>
<p :isRed:style="color:red">
```

More Embedded Logic

Attributes:

```
<input type=checkbox :isChecked:checked>
<p :isRed:style="color:red">
```

Maybe values:

```
$maybe name <- maybeName
  <p>Your name is #{name}
$nothing
  <p>I don't know your name.
```

More Embedded Logic

Attributes:

```
<input type=checkbox :isChecked:checked>
<p :isRed:style="color:red">
```

Maybe values:

```
$maybe name <- maybeName
  <p>Your name is #{name}
$nothing
  <p>I don't know your name.
```

```
$maybe Person firstName lastName <- maybePerson
  <p>Your name is #{firstName} #{lastName}
```

More Embedded Logic

Pattern matching:

```
$case foo
  $of Left bar
    <p>It was left: #{bar}
  $of Right baz
    <p>It was right: #{baz}
```

More Embedded Logic

Pattern matching:

```
$case foo
  $of Left bar
    <p>It was left: #{bar}
  $of Right baz
    <p>It was right: #{baz}
```

With:

```
$with foo <- (long ugly) expression that $ should only $ happen once
  <p>But I'm going to use #{foo} multiple times. #{foo}
```

Lucius

- Intended to be a superset of CSS

Lucius

- Intended to be a superset of CSS
- Nested blocks

Lucius

- Intended to be a superset of CSS
- Nested blocks
- Variable definitions

Lucius

- Intended to be a superset of CSS
- Nested blocks
- Variable definitions
- Mixins

Lucius

```
@textcolor: #494949;                              /* Variable definition */
body {
    color: #{textcolor};                          /* Variable interpolation */
    font-size: 16px;
}
.footer {
    background: url(@{BackgroundImageR}); /* URL interpolation */
    height: 150px;
    text-align: center;
    > .container {                                /* Nested block */
        padding: 45px 0 0 0;
    }
    p {
        margin-top: 45px;
        margin-bottom: 0px;
    }
}
```

Cassius

Just Lucius, but whitespace instead of brackets and semicolons.
You don't see this used too much.

```
@textcolor: #494949
body
    color: #{textcolor}
    font-size: 16px
.footer
    background-image: url(@{BackgroundImageR})
    height: 150px
    text-align: center
    > .container
        padding: 45px 0 0 0
    p
        margin-top: 45px
        margin-bottom: 0px
```

Julius

Simplist of the three

Julius

Simplist of the three
Performs no transformations other than allowing the forms of interpolation

Julius

Simplist of the three
Performs no transformations other than allowing the forms of interpolation

```
$(function(){
    $("section.#{sectionClass}").hide();
    $("#mybutton").click(function(){
        document.location = "@{SomeRouteR}";
    });
    ^{addBling}
});
```

QuasiQuoters

```
[hamlet| <h1>My Title |]
[lucius| h1 { color: green } |]
[julius| alert("Hi") |]
```

QuasiQuoters

```
[hamlet| <h1>My Title |]
[lucius| h1 { color: green } |]
[julius| alert("Hi") |]
```

Wait a minute, what was that whamlet Quasi-Quoter?

Widgets

Widgets

Yesod's solution to modular and composable web elements.
Each widget has these components:

- The title

Widgets

Yesod's solution to modular and composable web elements.
Each widget has these components:

- The title
- External stylesheets
- External Javascript

Widgets

Yesod's solution to modular and composable web elements.
Each widget has these components:

- The title
- External stylesheets
- External Javascript
- CSS declarations
- Javascript code

Widgets

Yesod's solution to modular and composable web elements.
Each widget has these components:

- The title
- External stylesheets
- External Javascript
- CSS declarations
- Javascript code
- Arbitrary <head> content
- Arbitrary <body> content

Widget Monad

Widgets can be composed monadically.

```
myWidget1 = do
  toWidget [hamlet|<h1>My Title|]
  toWidget [lucius|h1 { color: green } |]
```

Widget Monad

Widgets can be composed monadically.

```
myWidget1 = do
  toWidget [hamlet|<h1>My Title|]
  toWidget [lucius|h1 { color: green } |]

myWidget2 = do
  setTitle "My Page Title"
  addScriptRemote "http://www.example.com/script.js"
```

Widget Monad

Widgets can be composed monadically.

```
myWidget1 = do
  toWidget [hamlet|<h1>My Title|]
  toWidget [lucius|h1 { color: green } |]

myWidget2 = do
  setTitle "My Page Title"
  addScriptRemote "http://www.example.com/script.js"

myWidget = do
  myWidget1
  myWidget2
```

Widget Monad

Widgets can be composed monadically.

```
myWidget1 = do
  headerClass <- newIdent
  toWidget [hamlet| <h1 .#{headerClass}>My Title|]
  toWidget [lucius| .#{headerClass} { color: green } |]

myWidget2 = do
  setTitle "My Page Title"
  addScriptRemote "http://www.example.com/script.js"

myWidget = do
  myWidget1
  myWidget2
```

Whamlet

```
wrapper =
  [hamlet|
    <h1> My Site
    <div>
        ^{content}
  |]

content = [hamlet|
            <p> This is the content!
          |]
```

Whamlet

```
wrapper =
  [hamlet|
    <h1> My Site
    <div>
        ^{content}
  |]

content = do
  toWidget
    [lucius|
      p {
          text-decoration: underline;
      }
    |]
  toWidget
    [hamlet|
      <p> This is the content!
    |]
```

Whamlet

```
wrapper =
  [hamlet|
    <h1> My Site
    <div>
        ^{content} <!-- Type Error -->
  |]

content = do
  toWidget
    [lucius|
      p {
          text-decoration: underline;
      }
    |]
  toWidget
    [hamlet|
      <p> This is the content!
    |]
```

Whamlet

```
wrapper =
  [whamlet|
    <h1> My Site
    <div>
        ^{content}
  |]

content = do
  toWidget
    [lucius|
      p {
          text-decoration: underline;
      }
    |]
  toWidget
    [hamlet|
      <p> This is the content!
    |]
```

Exercise Two

Exercise Two

```
getHomeR = defaultLayout $ do
      setTitle "Welcome Home"
      addScriptRemote "https://code.jquery.com/jquery-3.2.1.js"
      toWidget
        [julius|
          $("h1").click(function(){
              alert("Header Clicked");
          }); |]

      toWidget [lucius| h1 { color: green } |]
      toWidgetHead
        [hamlet|<meta name=description content=#{contentDescription}>|]

      [whamlet|
         <h1>Welcome Back!
         ^{footer copyright}
      |]

    where contentDescription = "The homepage" :: String
          copyright = "Someone" :: String
```

Table of Contents

1. Basic Background
2. Set up Codio & Dive into an exercise
3. Content Creation
4. Handlers that actually do something
   - Getting data from the route
   - Handler Monad
   - Keeping global values
   - Exercise Three
5. Industrial Structure
6. Authentication and Authorization
7. Forms
8. Persistence

Getting data from the route

What mkYesod generates from the routes

```
mkYesod "App" [parseRoutes| /home HomeR GET |]
```

What mkYesod generates from the routes

```
mkYesod "App" [parseRoutes| /home HomeR GET |]
```

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

instance ParseRoute App where
    parseRoute (["home"], _) = Just HomeR
    parseRoute _             = Nothing

instance YesodDispatch App where
    yesodDispatch env req = yesodRunner handler env mroute req
      where
        mroute = parseRoute (pathInfo req, textQueryString req)
        handler =
            case mroute of
                Nothing -> notFound
                Just HomeR ->
                    case requestMethod req of
                        "GET" -> getHomeR
                        _     -> badMethod
```

Dynamic Routes

```
/person/#Text          PersonR GET POST
/month/#Text/day/#Int DateR   GET
```

Dynamic Routes

```
/person/#Text          PersonR GET POST
/month/#Text/day/#Int DateR   GET
```

```
<a href=@{PersonR "Gavin"}>Gavin
<a href=@{DateR "May" 25}>When is LambdaConf?
```

Dynamic Routes

```
/person/#Text          PersonR GET POST
/month/#Text/day/#Int DateR   GET
```

```
<a href=@{PersonR "Gavin"}>Gavin
<a href=@{DateR "May" 25}>When is LambdaConf?
```

```
getPersonR :: Text -> Handler Html
getPersonR name = undefined

getDateR :: Text -> Int -> Handler Html
getDateR month day = undefined
```

Handler Monad

Handler Monad

```
getHomeR :: Handler Html

type Handler a = HandlerT site IO a

getHomeR :: HandlerT Task IO Html
```

Handler Monad

```
getPersonR :: Text -> Handler Html
getPersonR name = do
    let reversedName = Data.Text.reverse name
    defaultLayout $
        [whamlet|
            <p> Name is: #{name}
            <p> Name reversed is: #{reversedName}
```

Some functions in the Handler Monad

```
-- | Returns the application's foundation value
getYesod :: Handler App
-- | returns a new identifier
newIdent :: Handler Text
-- | Redirect status 303
redirect :: Route App -> Handler a
-- | 404 Error
notFound :: Handler a
-- | 405 method not supported error
badMethod :: Handler a
-- | 401 not authenticated
notAuthenticated :: Handler a
-- | 403 Permission Denied
permissionDenied :: Handler a
```

IO Actions in Handlers

Because Handler is a monad transformer over IO, IO actions can be run with liftIO.

```
getHomeR :: Handler Html
getHomeR = do
    currentTime <- liftIO $ getCurrentTime
    defaultLayout $
      [whamlet|
        <p>It is currently: #{show currentTime}
      |]
```

Keeping global values

Foundation Datatype

```haskell
data Task = Task

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

instance Yesod Task

getHomeR :: Handler Html
getHomeR = do
    defaultLayout [whamlet|<p>This site is for: Lambdaconf|]

main :: IO ()
main = warp 3000 Task
```

Foundation Datatype

```
data App = App { appFor :: String }

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

instance Yesod App

getHomeR :: Handler Html
getHomeR = do
    who <- fmap appFor getYesod
    defaultLayout [whamlet|<p>This site is for: #{who}|]

main :: IO ()
main = warp 3000 (App { appFor = "PureScript Conf" })
```

Visitor Counter

```
data App = App { visitors :: IORef Int }

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

instance Yesod App

getHomeR :: Handler Html
getHomeR = do
    visitorsRef <- fmap visitors getYesod
    visitors <-
        liftIO $ atomicModifyIORef visitorsRef $ \i ->
        (i + 1, i + 1)
    defaultLayout
        [whamlet| <p>Welcome, you are visitor number #{visitors}. |]

main :: IO ()
main = do
    visitorsRef <- newIORef 0
    warp 3000 (App { visitors = visitorsRef })
```

Exercise Three

Exercise Three

In this exercise you'll be creating an app that contains a page which either adds or subtracts two numbers depending on configuration in the foundation datatype. You'll use dynamic routes to receive the Ints, and then the foundation datatype contains the binary operation to be performed on the numbers from the route.

```
data Task = Task { taskBinOp :: Int -> Int -> Int }
```

Table of Contents

1. Basic Background
2. Set up Codio & Dive into an exercise
3. Content Creation
4. Handlers that actually do something
5. Industrial Structure
   - Breaking up the code
   - Scaffolding Structure
   - Changing Default Behaivor
   - Sessions
   - Exercise Four
6. Authentication and Authorization
7. Forms
8. Persistence

Breaking up the code

Logically splitting up the Application

One issue with how we've been doing things so far, is that that if we keep putting everything in the same file, with lots of Quasi-Quotes all around, things become really ugly and unmaintainable.

```
getHomeR :: Handler Html
getHomeR = defaultLayout $ do
         setTitle "Welcome Home"
         [whamlet|<h1>Welcome Back!|]
```

Handlers need our Route definitions.

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

Yet mkYesod needs our Handlers to define dispatch over routes.

Splitting mkYesod

```
mkYesodData "Task" [parseRoutes|
/       HomeR GET
|]
```

This generates the route data type and the rendering functions, but doesn't define dispatch.

```
mkYesodDispatch "Task" resourcesTask
```

This generates the actual dispatch code.

Getting rid of Quasi-Quoters

```
mkYesodData "Task" [parseRoutes|
/       HomeR GET
|]
```

Instead of using Quasi-Quoters we want to put them into external files.

```
mkYesodData "Task" $(parseRoutesFile "routes")
```

Getting rid of Quasi-Quoters

```
getHomeR =
    defaultLayout $
        toWidget [hamlet|<p>This is the content!|]
```

Instead of using Quasi-Quoters we want to put them into external files.

```
getHomeR = defaultLayout $ toWidget $(hamletFile "home.hamlet")
```

Scaffolding Structure

Site Scaffolding

```
Project/
├── app/ ..................... General execution wrappers
├── config/ ................ Configuration loaded by the application
│   ├── favicon.ico
│   ├── robots.txt
│   ├── routes
│   └── settings.yml
├── Handler/ ..................... Files containing handlers
│   ├── Common.hs
│   └── Home.hs
├── Import/
├── templates/
├── Application.hs
├── Foundation.hs
├── Import.hs
└── Settings.hs
```

Site Scaffolding

```
Project/
├── app/ ..... General execution wrappers
├── config/ ..... Configuration loaded by the application
├── Handler/ ..... Files containing handlers
├── Import/
│   └── NoFoundation.hs
├── templates/
│   ├── default-layout-wrapper.hamlet
│   ├── default-layout.hamlet
│   ├── default-message-widget.hamlet
│   └── homepage.hamlet
├── Application.hs
├── Foundation.hs
├── Import.hs
└── Settings.hs
```

Scaffolding's Foundation

```
data App = App
  { appSettings     :: AppSettings
  , appLogger       :: Logger
  }
```

Scaffolding's Foundation

```
data App = App
  { appSettings     :: AppSettings
  , appLogger       :: Logger
  }
```

appSettings contains settings for the application, which are stored in a yaml file, config/settings.yml.

Scaffolding's Foundation

```
data App = App
  { appSettings     :: AppSettings
  , appLogger       :: Logger
  }
```

appSettings contains settings for the application, which are stored in a yaml file, config/settings.yml.

```
host:             "_env:HOST:*4" # any IPv4 host
port:             "_env:PORT:3000"
ip-from-header: "_env:IP_FROM_HEADER:false"
copyright: Insert copyright statement here
```

This is parsed using a parser in Settings.hs.

Scaffolding's Foundation

```
data App = App
  { appSettings     :: AppSettings
  , appLogger       :: Logger
  }
```

appLogger is the application's logger, which in general doesn't haven't to be interacted with directly. This is initialized in Application.hs, where the foundation value is constructed.

```
makeFoundation :: AppSettings -> IO App
makeFoundation appSettings = do
    -- Initialize logger
    appLogger <- newStdoutLoggerSet defaultBufSize >>= makeYesodLogger

    return $ App {..}
```

This is parsed using a parser in Settings.hs.

Changing Default Behaivor

defaultLayout?

What was that defaultLayout thing?

Yesod Typeclass

The Yesod typeclass gives a central place for defining settings for the application. Every method has intelligent defaults, so no implementation is required, however extensive modification is possible.

```
class RenderRoute site => Yesod site where
    approot :: Approot site
    errorHandler :: ErrorResponse -> Handler TypedContent
    defaultLayout :: Widget -> Handler Html
    isAuthorized :: Route site -> Bool -> Handler AuthResult
    authRoute :: site -> Maybe (Route site)
    makeSessionBackend :: site -> IO (Maybe SessionBackend)
    yesodMiddleware :: ToTypedContent res => Handler res -> Handler res
    defaultMessageWidget :: Html -> HtmlUrl (Route site) -> Widget
```

defaultLayout

```haskell
defaultLayout widget = do
  master <- getYesod
  mmsg <- getMessage

  -- We break up the default layout into two components:
  -- default-layout is the contents of the body tag, and
  -- default-layout-wrapper is the entire page.
  pc <- widgetToPageContent $(widgetFile "default-layout")
  withUrlRenderer
    $(hamletFile "templates/default-layout-wrapper.hamlet")
```

default-layout-wrapper.hamlet

```
$doctype 5
<html lang="en">
  <head>
    <meta charset="UTF-8">

    <title>#{pageTitle pc}
    <meta name="description" content="">
    <meta name="author" content="">

    <meta name="viewport" content="width=device-width,initial-scale=1">

    ^{pageHead pc}

  <body>
    ^{pageBody pc}
```

widgetFile?

This is a nice scaffolding feature that allows you to load a collection of files together as a widget.

For example:

```
$(widgetFile "homepage")
```

will attempt to read homepage.hamlet, homepage.lucius, and homepage.julius, and construct a widget out of them.

default-layout.hamlet

```
<div>
    <!-- Page Contents -->
    $maybe msg <- mmsg
        <div>#{msg}

    <div>
        ^{widget}

  <!-- Footer -->
  <footer>
      <div>
          <a href=@{HomeR}>Home
          <p> #{appCopyright $ appSettings master}
```

Sessions

Sessions

A session is a way to keep state about an interaction with a client. This is generally something that is desired to be minimized, but is fairly often unavoidable, such as a shopping cart implementation.

Sessions stored on the server are available, but not the default due to the additional overhead of doing additional database lookups to service requests.

Instead the approach usually used by Yesod is to store the session in a cookie on the client. This cookie is encrypted to prevent inspection and signed to prevent tampering.

Sessions

The client session is actually enabled in the default sessionBackend implementation, but the scaffolding we are using overrides it to change the location of the file that stores the encryption key for the cookies.

```
-- Store session data on the client in encrypted cookies,
-- default session idle timeout is 120 minutes
makeSessionBackend _ = Just <$> defaultClientSessionBackend
    120     -- timeout in minutes
    "config/client_session_key.aes"
```

Sessions

Sessions in Yesod are modeled as a basic key value store.

```
type SessionMap = Map Text ByteString
lookupSession :: MonadHandler m => Text -> m (Maybe Text)
lookupSessionBS :: MonadHandler m => Text -> m (Maybe ByteString)
getSession :: MonadHandler m => m SessionMap
setSession :: MonadHandler m => Text -> Text -> m ()
setSessionBS :: MonadHandler m => Text -> ByteString -> m ()
deleteSession :: MonadHandler m => Text -> m ()
clearSession :: MonadHandler m => m ()
```

Messages

One major use of sessions is messages. These solve a common case: the user performs a POST request, the web app takes the required action, and then wants to simultaneously redirect the user to a new page and display a message indicating the result of the action. (This is known as Post/Redirect/Get).

Yesod provides a pair of functions to make handling this workflow easy: setMessage stores a message in the session, and getMessage both reads this message and deletes it from the store to prevent it from being displayed again.

The common approach is to check for messages in defaultLayout so that the message can be shown to the user immediately, without having to add getMessage calls in every handler.

Messages

This is included in our scaffolding's defaultLayout.

```
defaultLayout widget = do
  master <- getYesod
  mmsg <- getMessage

  pc <- widgetToPageContent $(widgetFile "default-layout")
  withUrlRenderer
    $(hamletFile "templates/default-layout-wrapper.hamlet")
```

```
<div #wrapper>
  <div .container>
    $maybe msg <- mmsg
      <div #message>#{msg}

  <div .container>
    ^{widget}
```

Exercise Four

Exercise Four

In this exercise you'll start using the scaffolding. The task will be to add a new page that simply sets the a message in the session and redirect to the homepage. Then I'd like you to edit the default-layout.hamlet file (or if you'd rather add a default-layout.lucius file) and add some form of change or styling to how pages are displayed. Also add some message to the copyright field of the config/setting.yml.

Table of Contents

1. Basic Background
2. Set up Codio & Dive into an exercise
3. Content Creation
4. Handlers that actually do something
5. Industrial Structure
6. Authentication and Authorization
   - isAuthorized
   - yesod-auth
   - How do requests get routed to the Auth subsytem?
   - Exercise Five
7. Forms
8. Persistence

isAuthorized

isAuthorized

```
isAuthorized :: Route site -> Bool -> Handler AuthResult
isAuthorized HomeR _ = return Authorized
isAuthorized (PostR _) False = return Authorized
isAuthorized (PostR _) True =  isAuthenticated
isAuthorized (AdminR _) _ = isAdmin

isAuthenticated :: Handler AuthResult
isAuthenticated = do
    muid <- maybeAuthId
    return $ case muid of
        Nothing -> AuthenticationRequired
        Just _ -> Authorized

isAdmin :: Handler AuthResult
isAdmin = do
    muid <- maybeAuthId
    case muid of
      Nothing -> AuthenticationRequired
      Just "gavwhela@gmail.com" -> Authorized
      Just _ -> Unauthorized "Admin only page"
```

Authentication and Authorization

But what's providing maybeAuthId?
What happens when authentication is required?

yesod-auth

Yesod Auth Plugins

The yesod-auth package provides a unified interface for a number of available authencation plugins. The main requirement of a backend is that it identifies a user based on some unique string, such as a token, email, or username. Each plugin provides some mechanism for logging in. When a login is successful, the plugin sets a value in the user's session that indentifies them.

What yesod-auth provides

```
maybeAuthId :: Handler (Maybe (AuthId site))
maybeAuthPair :: Handler (Maybe (AuthId site, AuthEntity site))
maybeAuth :: Handler (Maybe (Entity val))
requireAuthId :: Handler (AuthId site)
requireAuthPair :: Handler (AuthId site, AuthEntity site)
requireAuth :: Handler (Entity val)
```

What yesod-auth provides

```
maybeAuthId :: Handler (Maybe (AuthId site))
maybeAuthPair :: Handler (Maybe (AuthId site, AuthEntity site))
maybeAuth :: Handler (Maybe (Entity val))
requireAuthId :: Handler (AuthId site)
requireAuthPair :: Handler (AuthId site, AuthEntity site)
requireAuth :: Handler (Entity val)
```

In order to use any of these, your foundation must be an instance of the YesodAuth typeclass

YesodAuth Typeclass

Minimal complete definition: loginDest, logoutDest, (authenticate|getAuthId), authPlugins, authHttpManager

```
class (Yesod master) => YesodAuth master where
    type AuthId master
    authLayout :: Widget -> Handler Html
    loginDest :: master -> Route master
    logoutDest :: master -> Route master
    authenticate :: Creds master -> Handler (AuthenticationResult master)
    getAuthId :: Creds master -> Handler (Maybe (AuthId master))
    authPlugins :: master -> [AuthPlugin master]
    loginHandler :: HandlerT Auth Handler Html
    redirectToReferer :: master -> Bool
    authHttpManager :: master -> Manager
    maybeAuthId :: Handler (Maybe (AuthId master))
```

YesodAuth Typeclass instance

```
instance YesodAuth App where
    type AuthId app = Text
    loginDest _ = HomeR
    logoutDest _ = HomeR
    redirectToReferer _ = True
    authenticate creds = return $ Authenticated $ credsIdent creds
    authPlugins _ = [authDummy]
    authHttpManager = error "No manager needed"
```

How do requests get routed to the Auth subsytem?

Routing to subsites

The Auth subsystem provides what's known as a subsite, which have their own internal dispatch, but can be included as a component of an application.

Routing to subsites

The Auth subsystem provides what's known as a subsite, which have their own internal dispatch, but can be included as a component of an application.

```
/auth    AuthR    Auth    getAuth
```

Routing to subsites

The Auth subsystem provides what's known as a subsite, which have their own internal dispatch, but can be included as a component of an application.

```
/auth    AuthR    Auth    getAuth
```

Then we want to tell our Yesod instance where to direct users to login.

```
authRoute _ = Just $ AuthR LoginR
```

Exercise Five

Exercise Five

In this exercise you'll be starting off with a scaffolded site which has a YesodAuth instance, with the Dummy auth plugin. The task is to add a page that requires being authenticated, and a page that requires being authenticated as a specific user. At least one of these should display the user's AuthId on the page. For here on we're also using a scaffolding with some default styling, and a new menu bar defined with haskell types in Foundation.hs

Table of Contents

1. Basic Background
2. Set up Codio & Dive into an exercise
3. Content Creation
4. Handlers that actually do something
5. Industrial Structure
6. Authentication and Authorization
7. Forms
   - Fields
   - Applicative Forms
   - Building up
   - Running Forms
   - Final Notes
   - Exercise Six
8. Persistence

Intro

The yesod-form package gives a framework for building up reusable and composable forms from constituent parts.

Intro

The yesod-form package gives a framework for building up reusable and composable forms from constituent parts.

In order to use them, this is a required instance implementation:

```
instance RenderMessage App FormMessage where
    renderMessage _ _ = defaultFormMessage
```

Fields

Fields

Each field defines a way of parsing input from the user into a Haskell value, and how to create a widget to display the field.

```
textField :: Field Handler Text
passwordField :: Field Handler Text
textareaField :: Field Handler Textarea
hiddenField :: PathPiece p => Field Handler p
intField :: Integral i => Field Handler i
dayField :: Field Handler Day
htmlField :: Field Handler Html
emailField :: Field Handler Text
```

Field Validation

yesod-form provides convenience field transformers to add validation conditions to a field.

```
check :: (a -> Either Text a) -> Field Handler a -> Field Handler a
checkBool :: (a -> Bool) -> Text -> Field Handler a -> Field Handler a
checkM :: (a -> Handler (Either Text a)) ->
          Field Handler a -> Field Handler a
```

Field Validation

```haskell
nameField :: Field Handler Text
nameField = check validateName textField
-- Bad validation
validateName name
  | (length name) < 4 = Left "Name is too short"
  | otherwise = Right name
```

Field Validation

```haskell
nameField :: Field Handler Text
nameField = check validateName textField
-- Bad validation
validateName name
  | (length name) < 4 = Left "Name is too short"
  | otherwise = Right name
```

```haskell
-- Bad validation
nameField :: Field Handler Text
nameField = checkBool ((> 3) . length) errorMessage textField
errorMessage = "Name is too short"
```

Applicative Forms

Field to Applicative Form

A field can be converted into an applicative form using either areq or aopt. Both of these functions also take a FieldSettings, to customize the field, as well as a Maybe value, which if isJust, contains an initial value for the field.

Field to Applicative Form

```
areq :: Field Handler a -> FieldSettings -> Maybe a ->
        AForm Handler a
aopt :: Field Handler a -> FieldSettings -> Maybe (Maybe a) ->
        AForm Handler (Maybe a)
```

Field to Applicative Form

```
areq :: Field Handler a -> FieldSettings -> Maybe a ->
        AForm Handler a
aopt :: Field Handler a -> FieldSettings -> Maybe (Maybe a) ->
        AForm Handler (Maybe a)
```

```
textForm :: AForm Handler Text
textForm = areq textField "text" Nothing

mtextForm :: AForm Handler (Maybe Text)
mtextForm = aopt textField "mtext" Nothing
```

Field to Applicative Form

```
areq :: Field Handler a -> FieldSettings -> Maybe a ->
        AForm Handler a
aopt :: Field Handler a -> FieldSettings -> Maybe (Maybe a) ->
        AForm Handler (Maybe a)
```

```
textForm :: AForm Handler Text
textForm = areq textField "text" Nothing

mtextForm :: AForm Handler (Maybe Text)
mtextForm = aopt textField "mtext" Nothing
```

```
defaultTextForm :: Text -> AForm Handler Text
defaultTextForm defText = areq textField "text" (Just defText)

defaultMtextForm :: (Maybe Text) -> AForm Handler (Maybe Text)
defaultMtextForm defMtext = aopt textField "mtext" (Just defMtext)
```

FieldSettings

```
data FieldSettings = FieldSettings
         { fsLabel :: Text
         , fsTooltip :: Maybe Text
         , fsId :: Maybe Text
         , fsName :: Maybe Text
         , fsAttrs :: [(Text, Text)]
         }
```

FieldSettings

```
data FieldSettings = FieldSettings
         { fsLabel :: Text
         , fsTooltip :: Maybe Text
         , fsId :: Maybe Text
         , fsName :: Maybe Text
         , fsAttrs :: [(Text, Text)]
         }
```

Wait a minute, weren't we using a literal string as a FieldSettings? What's going on?

FieldSettings

```
data FieldSettings = FieldSettings
         { fsLabel :: Text
         , fsTooltip :: Maybe Text
         , fsId :: Maybe Text
         , fsName :: Maybe Text
         , fsAttrs :: [(Text, Text)]
         }
```

Wait a minute, weren't we using a literal string as a FieldSettings? What's going on?

```
instance IsString (FieldSettings a) where
  fromString s =
    FieldSettings (fromString s) Nothing Nothing Nothing []
```

FieldSettings

```haskell
nameForm :: AForm Handler Text
nameForm = areq textField nameFieldSettings Nothing
  where nameFieldSettings =
    FieldSettings
      { fsLabel = "Name"
      , fsTooltip = Just "Enter your name here"
      , fsId = Nothing
      , fsName = Nothing
      , fsAttrs = [("class", "namefield")]
      }
```

Modifying Form's Returned Value

```
sillyForm :: AForm Handler Int
sillyForm = ((+) 100) <$> areq intField "Num" Nothing
```

This just demonstrates that you can fmap a function over the value input into the field. This is a little troublesome however, because if you save the value for a default next time, you'll have to make sure to subtract the 100 again so it's not 100 more than what was originally entered.

Building up

Composing Applicative Forms

```
data Person = Person
    { personName    :: Text
    , personEmail   :: Text
    , personWebsite :: Maybe Text }

personForm :: AForm Handler Person
personForm = Person
  <$> areq textField "Name" Nothing
  <*> areq emailField "Email Address" Nothing
  <*> aopt textField "Website" Nothing
```

Non-Input Values

```
data Post = Post
    { postTitle    :: Text
    , postContents :: Textarea
    , postAuthor   :: UserId
    , postVal      :: Text
    , postCreated  :: UTCTime }

postForm :: UserId -> AForm Handler Post
postForm userId = Post
  <$> areq textField "Title" Nothing
  <*> areq textareaField "Contents" Nothing
  <*> pure userId
  <*> lift (fmap appGetFormVal getYesod)
  <*> lift (liftIO getCurrentTime)
```

Rendering Forms

How does the widgets for the fields of a form get composed together?

Rendering Forms

How does the widgets for the fields of a form get composed together?

```
type Form a = Html -> MForm Handler (FormResult a, Enctype)
type FormRender m a = AForm m a -> Form a

renderDivs :: FormRender m a
renderDivsNoLabels :: FormRender m a
renderTable :: FormRender m a
```

How forms usually look

```haskell
data Post = Post
    { postTitle    :: Text
    , postContents :: Textarea
    , postAuthor   :: UserId
    , postVal      :: Text
    , postCreated  :: UTCTime }

postForm :: UserId -> Form Post
postForm userId = renderDivs $ Post
  <$> areq textField "Title" Nothing
  <*> areq textareaField "Contents" Nothing
  <*> pure userId
  <*> lift (fmap appGetFormVal getYesod)
  <*> lift (liftIO getCurrentTime)
```

Running Forms

Running Forms

```
data FormResult a = FormMissing
                  | FormFailure [Text]
                  | FormSuccess a

runFormPost :: Form a -> Handler ((FormResult a, xml), Enctype)
runFormGet :: Form a -> Handler ((FormResult a, xml), Enctype)
runFormPostNoToken :: Form a -> Handler ((FormResult a, xml), Enctype)
generateFormPost :: Form a -> Handler (xml, Enctype)
generateFormGet' :: Form a -> Handler (xml, Enctype)
```

Example

```
postNewPostR :: Handler Html
postNewPostR = do
  user <- requireAuthId
  ((res, formWidget), enctype) <- runFormPost $ postForm user
  case res of
    FormSuccess entry -> do
        runDB $ insert_ entry
        setMessage "Successfully created post"
        redirect $ PostsR user
    _ -> defaultLayout $ do
              setTitle "New Post"
              $(widgetFile "new-post")

getNewPostR :: Handler Html
getNewPostR = do
  user <- requireAuthId
  (formWidget, enctype) <- generateFormPost $ postForm user
  defaultLayout $ do
      setTitle "New Post"
      $(widgetFile "new-post")
```

Example

```haskell
postNewPostR :: Handler Html
postNewPostR = do
  user <- requireAuthId
  ((res, formWidget), enctype) <- runFormPost $ postForm user
  case res of
    FormSuccess entry -> do
        runDB $ insert_ entry
        setMessage "Successfully created post"
        redirect $ PostsR user
    _ -> defaultLayout $ do
              setTitle "New Post"
              $(widgetFile "new-post")

getNewPostR :: Handler Html
getNewPostR = postNewPostR
```

Example

```haskell
postNewPostR :: Handler Html
postNewPostR = do
  user <- requireAuthId
  ((res, formWidget), enctype) <- runFormPost $ postForm user
  case res of
    FormSuccess entry -> do
        runDB $ insert_ entry
        setMessage "Successfully created post"
        redirect $ PostsR user
    _ -> defaultLayout $ do
              setTitle "New Post"
              $(widgetFile "new-post")

getNewPostR :: Handler Html
getNewPostR = postNewPostR
```

```
<form method=post action=@{NewPostR} enctype=#{enctype}>
  ^{formWidget}
  <button>Submit
```

Final Notes

Multiple Forms

Sometimes you'd like to have a single handler be able to run multiple forms, while identifying which form was submitted. The form transformer "identifyForm" is used to add a hidden form onto an existing form in order to identify it.

```
((fooRes, fooWidget), fooEnctype) <- runFormPost fooForm
((barRes, barWidget), barEnctype) <- runFormPost barForm
```

Becomes:

```
((fooRes, fooWidget), fooEnctype) <-
    runFormPost $ identifyForm "foo" fooForm
((barRes, barWidget), barEnctype) <-
    runFormPost $ identifyForm "bar" barForm
```

Other Types of Forms

There are two other types of forms available, monadic forms and input forms. Monadic forms have you create the final monadic form yourself, instead of having a FormRender function create it from an applicative form. Input forms basically are the same as applicative forms, but throw away the builtin widgets, and receive input from a form constructed elsewhere (in javascript, or in manual Html that was pre-existing).

Exercise Six

Exercise Six

For this exercise, you'll create a page with a form that creates a Post value containing title, author, and content fields. The author will be filled in from the auth value (we'll still be using dummy auth), so the page with the form will need to require auth. On a successful post of the form, a page without the form should be displayed, containing the content from the form.

What next?

We'd sure like somewhere to put the data from these forms...

Table of Contents

1. Basic Background
2. Set up Codio & Dive into an exercise
3. Content Creation
4. Handlers that actually do something
5. Industrial Structure
6. Authentication and Authorization
7. Forms
8. Persistence
   - Entity Definition DSL
   - Actions
   - Connecting to Yesod
   - Exercise Seven

Features

Persistent is a Haskell library for handling data storage interactions.

- Not restricted to Yesod, general purpose.

Features

Persistent is a Haskell library for handling data storage interactions.

- Not restricted to Yesod, general purpose.
- Database agnostic, Support for PostgreSQL, SQLite, MySQL, and MongoDB.

Features

Persistent is a Haskell library for handling data storage interactions.

- Not restricted to Yesod, general purpose.
- Database agnostic, Support for PostgreSQL, SQLite, MySQL, and MongoDB.
- Handles marshalling of Haskell datatypes into the storage layer.

Features

Persistent is a Haskell library for handling data storage interactions.

- Not restricted to Yesod, general purpose.
- Database agnostic, Support for PostgreSQL, SQLite, MySQL, and MongoDB.
- Handles marshalling of Haskell datatypes into the storage layer.
- Type safe, concise, declarative syntax.

Features

Persistent is a Haskell library for handling data storage interactions.

- Not restricted to Yesod, general purpose.
- Database agnostic, Support for PostgreSQL, SQLite, MySQL, and MongoDB.
- Handles marshalling of Haskell datatypes into the storage layer.
- Type safe, concise, declarative syntax.
- Automatic database migration system.

Data Definition Differences

```
data Person = Person
    { personName :: Text
    , personAge  :: Int
    }
```

Data Definition Differences

```haskell
data Person = Person
    { personName :: Text
    , personAge  :: Int
    }
```

```sql
CREATE TABLE person(id SERIAL PRIMARY KEY,
                    name VARCHAR NOT NULL,
                    age INTEGER);
```

Entity Definition DSL

Entity Definition DSL

```
Person
    name String
    age Int
    deriving Show
```

Generated Code

```haskell
data Person = Person
  { personName :: !String
  , personAge :: !Int
  }
deriving Show

type PersonId = Key Person

instance PersistEntity Person where
    newtype Key Person = PersonKey (BackendKey SqlBackend)
        deriving (PersistField, Show, Eq, Read, Ord)
    -- A GADT allows fields to be matched with their datatypes.
    data EntityField Person typ where
        PersonId   :: EntityField Person PersonId
        PersonName :: EntityField Person String
        PersonAge  :: EntityField Person Int

    data Unique Person
    type PersistEntityBackend Person = SqlBackend
    -- ...
```

Actions

Extended Entities for Examples

```
Person
    name String
    age Int Maybe
    deriving Show
BlogPost
    title String
    authorId PersonId
    deriving Show
```

Actions

```
johnId <- insert $ Person "John Doe" $ Just 35
janeId <- insert $ Person "Jane Doe" Nothing
```

Actions

```
johnId <- insert $ Person "John Doe" $ Just 35
janeId <- insert $ Person "Jane Doe" Nothing

insert $ BlogPost "My fr1st p0st" johnId
insert $ BlogPost "One more for good measure" johnId
```

Actions

```
johnId <- insert $ Person "John Doe" $ Just 35
janeId <- insert $ Person "Jane Doe" Nothing

insert $ BlogPost "My fr1st p0st" johnId
insert $ BlogPost "One more for good measure" johnId

oneJohnPost <- selectList [BlogPostAuthorId ==. johnId] [LimitTo 1]
liftIO $ print (oneJohnPost :: [Entity BlogPost])
```

Actions

```
johnId <- insert $ Person "John Doe" $ Just 35
janeId <- insert $ Person "Jane Doe" Nothing

insert $ BlogPost "My fr1st p0st" johnId
insert $ BlogPost "One more for good measure" johnId

oneJohnPost <- selectList [BlogPostAuthorId ==. johnId] [LimitTo 1]
liftIO $ print (oneJohnPost :: [Entity BlogPost])

john <- get johnId
liftIO $ print (john :: Maybe Person)
```

Actions

```
johnId <- insert $ Person "John Doe" $ Just 35
janeId <- insert $ Person "Jane Doe" Nothing

insert $ BlogPost "My fr1st p0st" johnId
insert $ BlogPost "One more for good measure" johnId

oneJohnPost <- selectList [BlogPostAuthorId ==. johnId] [LimitTo 1]
liftIO $ print (oneJohnPost :: [Entity BlogPost])

john <- get johnId
liftIO $ print (john :: Maybe Person)

update janeId [PersonAge =. Just 33]
```

Actions

```
johnId <- insert $ Person "John Doe" $ Just 35
janeId <- insert $ Person "Jane Doe" Nothing

insert $ BlogPost "My fr1st p0st" johnId
insert $ BlogPost "One more for good measure" johnId

oneJohnPost <- selectList [BlogPostAuthorId ==. johnId] [LimitTo 1]
liftIO $ print (oneJohnPost :: [Entity BlogPost])

john <- get johnId
liftIO $ print (john :: Maybe Person)

update janeId [PersonAge =. Just 33]

delete janeId
deleteWhere [BlogPostAuthorId ==. johnId]
```

More About Selects

```
Person
    firstName String
    lastName  String
    age       Int
```

More About Selects

```
Person
    firstName String
    lastName  String
    age       Int
```

```
people <- selectList [ PersonAge >. 25, PersonAge <=. 30 ] []
people <- selectList (    [ PersonAge >=. 25, PersonAge <=. 30 ]
                       ||. [ PersonAge >. 35,  PersonAge <. 40 ] ) []
people <- selectList [ PersonFirstName ==. "Gavin" ] []
people <- selectList [ PersonFirstName <-. ["Gavin", "Alex"]] []
people <- selectList [ PersonFirstName /<-. ["James", "Albert"]] []

personEntity <- selectFirst [ PersonFirstName ==. "Gavin" ]
case personEntity of
  Nothing -> undefined
  Just (Entity personId person) -> undefined
```

Sorting, Limits, and Offsets

```
resultsForPage pageNumber = do
  let resultsPerPage = 10
  selectList
      [ PersonAge >=. 18 ]
      [ Desc PersonAge
      , Asc PersonLastName
      , Asc PersonFirstName
      , LimitTo resultsPerPage
      , OffsetBy $ (pageNumber - 1) * resultsPerPage ]
```

Updating Records

```
personId <- insert $ Person "Gavin" "Whelan" 21
update personId [ PersonAge =. 22 ]
update personId [ PersonAge +=. 1 ]
updateWhere [PersonFirstName ==. Gavin] [ PersonAge -=. 1 ]
replace personId $ Person "John" "Doe" 27
```

Deleting Records

```
personId <- insert $ Person "Gavin" "Whelan" 21
delete personId
deleteWhere [ PersonFirstName ==. "Gavin" ]
deleteWhere ([] :: [Filter Person])
```

Uniqueness Constraints

```
Person
    firstName String
    lastName String
    age Int
    PersonName firstName lastName
    deriving Show
```

Uniqueness Constraints

```
Person
    firstName String
    lastName String
    age Int
    PersonName firstName lastName
    deriving Show
```

```
insert_ $ Person "Gavin" "Whelan" 1
gavin <- getBy $ PersonName "Gavin" "Whelan"
liftIO $ print gavin
deleteBy $ PersonName "Gavin" "Whelan"
```

Relationships

```
Person
  name String
Store
    name String
PersonStore
    personId PersonId
    storeId StoreId
    UniquePersonStore personId storeId
```

Relationships

```
Person
  name String
Store
    name String
PersonStore
    personId PersonId
    storeId StoreId
    UniquePersonStore personId storeId
```

```
bruce <- insert $ Person "Bruce Wayne"
michael <- insert $ Person "Michael"

target <- insert $ Store "Target"
gucci <- insert $ Store "Gucci"
sevenEleven <- insert $ Store "7-11"

insert $ PersonStore bruce gucci
insert $ PersonStore bruce sevenEleven
insert $ PersonStore michael target
insert $ PersonStore michael sevenEleven
```

Connecting to Yesod

How Entity Definitions Are Loaded

config/models:

```
Person
    firstName String
    lastName String
    age Int
    PersonName firstName lastName
    deriving Show
```

Model.hs:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
    $(persistFileWith lowerCaseSettings "config/models")
```

Adding Instances

Foundation.hs:

```
data App = App
    { appSettings    :: AppSettings
    , appConnPool    :: ConnectionPool -- ^ Database connection pool.
    , appLogger      :: Logger
    }
```

Adding Instances

Foundation.hs:

```haskell
data App = App
    { appSettings    :: AppSettings
    , appConnPool    :: ConnectionPool -- ^ Database connection pool.
    , appLogger      :: Logger
    }
```

```haskell
-- How to run database actions.
instance YesodPersist App where
    type YesodPersistBackend App = SqlBackend
    runDB action = do
        master <- getYesod
        runSqlPool action $ appConnPool master
```

Adding Instances

Foundation.hs:

```
data App = App
    { appSettings    :: AppSettings
    , appConnPool    :: ConnectionPool -- ^ Database connection pool.
    , appLogger      :: Logger
    }
```

```
-- How to run database actions.
instance YesodPersist App where
    type YesodPersistBackend App = SqlBackend
    runDB action = do
        master <- getYesod
        runSqlPool action $ appConnPool master
```

```
instance YesodPersistRunner App where
    getDBRunner = defaultGetDBRunner appConnPool
```

Database Connection

config/settings.yml:

```
database:
  user:     "_env:PGUSER:lambdaconf"
  password: "_env:PGPASS:'2017'"
  host:     "_env:PGHOST:localhost"
  port:     "_env:PGPORT:5432"
  database: "_env:PGDATABASE:lambdaconf"
  poolsize: "_env:PGPOOLSIZE:10"
```

makeFoundation, Application.hs:

```
-- Create the database connection pool
pool <- flip runLoggingT logFunc $ createPostgresqlPool
    (pgConnStr  $ appDatabaseConf appSettings)
    (pgPoolSize $ appDatabaseConf appSettings)

-- Perform database migration using our application's logging settings.
runLoggingT (runSqlPool (runMigration migrateAll) pool) logFunc
```

Using Persistent in Handlers

```
/ HomeR GET
/person/#PersonId PersonR GET
```

```
getPersonR :: PersonId -> Handler Html
getPersonR personId = do
    person <- runDB $ get404 personId
    defaultLayout $
      [whamlet|
        <p>#{personFirstName person}
           #{personLastName person}
           is #{personAge person}
      |]
```

Generating Listings

```
-- List all people in the database
getHomeR :: Handler Html
getHomeR = do
  people <- runDB $ selectList [] [Asc PersonAge]
  defaultLayout $
    [whamlet|
      <ul>
        $forall Entity personid person <- people
          <li>
            <a href=@{PersonR personid}>#{personFirstName person}
    |]
```

Exercise Seven

Exercise Seven

For this exercise, we'll be finally doing something that is starting to look like a blog. You'll be starting with the form from the last exercise, but you'll create a post entity with title, author, and content fields instead of a haskell datatype. Then, instead of just displaying the posted data, you'll insert it in the database. You'll also need a handler to display an individual post, and the home page will give a listing with each entry providing a link to the individual post.

Starting a Yesod Project IRL

Available templates:

- yesod-minimal
- yesod-mongo
- yesod-mysql
- yesod-postgres
- yesod-postgres-fay
- yesod-simple
- yesod-sqlite

```
stack new my-project <template> && cd my-project
stack build yesod-bin cabal-install --install-ghc
stack build
stack exec -- yesod devel
```

https://www.yesodweb.com/page/quickstart

Thanks!