Introduction to Yesod

James Parker
jp@jamesparker.me

Example code

- Example code if you want to follow along:
  - https://github.com/jprider63/yesod-introduction

Yesod

- Web development framework for Haskell
- Alternatives: Happstack, Snap, Servant, etc

Why Yesod?

- Haskell
  - Strong static type system
  - Less bugs
  - Access to many packages on hackage
- Helps prevent most security vulnerabilities
  - CSRF, SQL injection, XSS
- Performant
- Production ready
  - https://builditbreakit.org/
  - https://pkauth.com/

Getting Started

Create a new project:

$ stack new project-name yesod-postgresql && cd project-name

Build project:

$ stack build

Getting Started

Configure your database settings (config/settings.yml):

```
database:
  user:     "_env:PGUSER:yesod-introduction_LOWER"
  password: "_env:PGPASS:yourpassword"
  host:     "_env:PGHOST:localhost"
  port:     "_env:PGPORT:5432"
  database: "_env:PGDATABASE:yesod-introduction_LOWER"
  poolsize: "_env:PGPOOLSIZE:10"
```

Getting Started

Set up database if needed:

```
CREATE ROLE "yesod-introduction_LOWER" PASSWORD 'yourpassword';
CREATE DATABASE "yesod-introduction_LOWER" OWNER "yesod-introduction_LOWER" ENCODING 'UTF8';
ALTER ROLE "yesod-introduction_LOWER" WITH LOGIN;
SET timezone='UTC';
```

- We’re using PostgreSQL
- Use psql to connect to PostgreSQL repl
- You may need edit other database settings to allow password authentication, ports, etc

Getting Started

Run in development mode:

$ stack exec -- yesod devel

Visit website:

http://localhost:3000/

Getting Started

Run in development mode:

$ stack exec -- yesod devel

- Changes made to source files should cause the website to be rebuilt
- Sometimes it doesn’t detect changes, so you need to force updates (ie. `touch src/Settings/StaticFiles.hs`)
- Sometimes development mode just doesn’t work, so rebuild manually (stack build --fast --flag yesod-introduction:dev)

Project Components

- Route parsing (config/routes)
- Handlers (src/Handler/*)
  - Code that generates responses to different requests
- Database schema (config/models)
- Static files (static/*)
- Common functionality (src/Foundation.hs)
  - Shared HTML+CSS layout
  - Authentication plugins

Route Parsing

- Yesod uses a domain specific language (DSL) to define routes
  - At compile time, Yesod uses Template Haskell to convert the DSL into Haskell code that parses the routes received in HTTP requests
- Advantages
  - Easier than writing your own parser
  - Prevents mistakes
  - Prevents dead links

Route Parsing

Edit config/routes:

/ HomeR GET POST

/profile/#Text ProfileR GET

Route Parsing

Edit config/routes:

/ HomeR GET POST

/profile/#Text ProfileR GET

- Specifies the route path

Route Parsing

Edit config/routes:

/ HomeR GET POST

/profile/#Text ProfileR GET

- Values are automatically parsed in paths
- Any type that implements the PathPiece typeclass can be included in a path

Route Parsing

Edit config/routes:

/ HomeR GET POST

/profile/#Text ProfileR GET

- Specifies which HTTP verbs are supported

Route Parsing

Edit config/routes:

/ HomeR GET POST

/profile/#Text ProfileR GET

- Specifies which handler function should be called when a path is parsed
  - getHomeR :: Handler Html
  - postHomeR :: Handler Html
  - getProfileR :: Text -> Handler Html

Route Parsing

Edit config/routes:

```
/ HomeR GET POST

/profile/#Text ProfileR GET
```

Yesod automatically creates a data type to represent routes:

```
data MyRoute =
    HomeR
  | ProfileR Text
```

- Used to link to different pages
- Type checker prevents dead links at compile time

Handlers

- Functions that generate responses for specific routes
- Run in the Handler (and Widget) monads
- General haskell code, but usually:
  - Render HTML, CSS, and JS
  - Set titles, set headers, send redirects
  - Run database queries
  - Send emails
  - Parse forms

Handlers

```
getHomeR :: Handler Html
getHomeR = defaultLayout [whamlet|
        <h1>
            Hello World!
    |]
```

- Basic handler that prints “Hello World!” inside a header
- Yesod uses a DSL to generate HTML

Handlers

```
getProfileR :: Text -> Handler Html
getProfileR username = defaultLayout $ do
    setTitle $ username <> " Profile"
    [whamlet|
        <a href="@{HomeR}">
            Homepage
        <h1>
            Profile: #{username}
    |]
```

- Argument received from route path
- Set web page title
- Link to homepage
- Include `username` in HTML

Handlers

- To create a new handler:
  - Create a module for your handler and implement your code
  - Add a route to config/routes
  - Import the new handler module in src/Application.hs
  - Add the module to your cabal file

Database Schema

Database schemas are specified as a DSL in config/models:

```
User
    username Text
    email Text Maybe
    UniqueUser username
```

- Table name
- Column names
- Column types
- Secondary key with uniqueness constraint

Database Schema

Database schemas are specified as a DSL in config/models:

```
User
    username Text
    email Text Maybe
    UniqueUser username
```

- Yesod automatically performs (safe) schema setup and migrations at startup

Database Schema

Database schemas are specified as a DSL in config/models:

```
User
    username Text
    email Text Maybe
    UniqueUser username
```

Corresponding Haskell data types are automatically created:

```
data User = {
    userUsername :: Text
  , userEmail :: Maybe Text
  }
```

Databases

- There are two main database libraries
  - persistent - basic queries
  - esqueleto - advanced queries with joins

Databases

```
getProfileR :: Text -> Handler Html
getProfileR username = do
    (Entity userId user) <- runDB $ getBy404 $ UniqueUser username
    let email = maybe “-” id $ userEmail user
    defaultLayout $ do
        setTitle $ username <> “ Profile”
        [whamlet|
            <a href=“@{HomeR}”>
                Homepage
            <h1>
                Profile: #{username}
                Email: #{email}
        |]
```

- Retrieve the user with the given username from the database
    - Returns a 404 not found error page if the username doesn’t exist
- Returns the user and the user’s primary key

Foundation.hs

- General website settings and functionality
- Define default HTML template in `defaultLayout`
- Setup authentication
  - Various authentication plugins are available
    - Password logins, OpenId, email challenges, dummy logins (for development)

Authentication

- maybeAuthId :: Handler (Maybe UserId)
  - Returns the primary key of the user if they are logged in
  - maybeAuth is a variant that returns the primary key and the user
- requireAuthId :: Handler UserId
  - Returns the primary key of the user if they are logged in
  - Redirects to the login page if they’re not
  - requireAuth also returns the primary key and the user

Forms

- Parse form data into Haskell data types
- Validate received data
- Generate HTML for displaying form
- Automatically insert CSRF tokens

Forms

Create a form for user registration:

```
registerForm :: Form User
registerForm = renderBootstrap3 BootstrapBasicForm $ User
    <$> areq textField (bfs ("Username" :: Text)) Nothing
    <*> aopt emailField (bfs ("Email" :: Text)) Nothing
```

- Returns a User
- Field label
- Required field
- Optional field
- Optional default value for fields

Forms

Require additional validation for username field:

```
registerForm :: Form User
registerForm = renderBootstrap3 BootstrapBasicForm $ User
    <$> areq usernameField (bfs ("Username" :: Text)) Nothing
    <*> aopt emailField (bfs ("Email" :: Text)) Nothing

    where
        usernameField = check validateUsername textField

        validateUsername u | Text.all isAlphaNum u = Right u
        validateUsername _ = Left ("Usernames must be alphanumeric." :: Text)
```

Forms

Render forms:

```
generateHtml :: Widget -> Enctype -> Handler Html
generateHtml form enctype = defaultLayout $ do
    [whamlet|
        <form .form-basic role=form method=post action="@{RegisterR}" enctype=#{enctype}>
            ^{form}
            <div .form-group>
                <button .btn .btn-primary .btn-lg .btn-block type="submit">
                    Register
    |]

getRegisterR :: Handler Html
getRegisterR = do
    (form, enctype) <- generateFormPost registerForm
    generateHtml form enctype
```

Forms

Process forms:

```
postRegisterR :: Handler Html
postRegisterR = do
    ((res, form), enctype) <- runFormPost registerForm
    case res of
        FormSuccess user@(User _username _email) -> do
            runDB $ insert_ user

            setMessage "Signed up!"

            redirect HomeR
        _ -> do
            -- Error case.
            generateHtml form enctype
```

Advanced

- Build REST APIs
- Use reflex + GHCJS to build fancy javascript applications
- Yesod is a combination of a lot of tools to make web development easier. You can pick and choose which you want to use.

Resources

- https://github.com/jprider63/yesod-introduction
- https://www.yesodweb.com/book
- https://groups.google.com/forum/#!forum/yesodweb
- http://hayoo.fh-wedel.de/
- https://www.haskell.org/hoogle/
- https://en.wikipedia.org/wiki/Post/Redirect/Get

Add functionality together?

- List all users?
- Users can follow other users?
- Other ideas?