Web Development

Passport, Passport-Local, Passport-local-mongoose and Express-Session

User Authentication

- Usually it is an in depth topic
- Requires a lot of concepts therefore, a lot of code
- Due to it this course's time constraint,
- We will use some packages to help us understand the concepts better and simplify the process.
  - Passport
  - Passport – local
  - Passport- express – local
  - Express-session

Passport

- Passport is Express-compatible authentication middleware for Node.js.
- Passport's sole purpose is to authenticate requests,
- Using an extensible set of plugins known as ***strategies***.
- Does not mount routes or assume any particular database schema,
- So you provide it how user data is stored by writing your data schema and storing user login and password information in the database.
- You provide Passport a request to authenticate, and Passport provides hooks for controlling what occurs when authentication succeeds or fails.

```
$ npm install passport
```

Passport - Strategies

- Passport uses the concept of strategies to authenticate requests.
- Strategies can range from
  - verifying username and password credentials,
  - delegated authentication using OAuth (for example, via Facebook or Twitter),
  - or federated authentication using OpenID.
- Before authenticating requests, the strategy (or strategies) used by an application must be configured.
- There are 480+ strategies.
- Find the ones you want at: passportjs.org

Passport - Sessions

- Passport will maintain persistent login sessions.
- In order for persistent sessions to work, the authenticated user must be serialized to the session, and deserialized when subsequent requests are made.
- Passport does not impose any restrictions on how your user records are stored.
- Instead, you provide functions to Passport which implements the necessary serialization and deserialization logic.

Passport – Middleware (express-session)

- To use Passport in an Express or Connect-based application, configure it with the required passport.initialize() middleware.
- If your application uses persistent login sessions,passport.session() middleware must also be used.
- The middlware we will be using is Express-Session to handle Passport Session.
- Passport provides an **authenticate()** function, which is used as route middleware to authenticate requests.

express-session

npm v1.17.1 | downloads 3.7M/month | travis passing | coverage 100%

Installation

This is a Node.js module available through the npm registry. Installation is done using the `npm install` command:

```
$ npm install express-session
```

API

```
var session = require('express-session')
```

```
app.use(require('express-session')({ secret: 'keyboard cat', resave: true, saveUninitialized: true }));
app.use(passport.initialize());
app.use(passport.session());
```

```
app.post('/login',
  passport.authenticate('local', { failureRedirect: '/login' }),
  function(req, res) {
    res.redirect('/');
  });
```

Passport -local

- Passport strategy for authenticating with a username and password.
- This module lets you authenticate using a username and password in your Node.js applications.
- By plugging into Passport, local authentication can be easily and unobtrusively integrated into any application or framework that supports Connect-style middleware, including Express.

```
$ npm install passport-local
```

```
var LocalStrategy = require("passport-local")
```

Passport –local - Mongoose

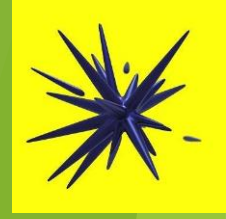
- Passport-Local Mongoose is a Mongoose plugin that simplifies building username and password login with Passport.
- You need to plugin passport-local-mongoose into your user schema while creating the user Model.
- Passport-Local Mongoose supports this setup by implementing a LocalStrategy and serializeUser/deserializeUser functions.

```
> npm install passport-local-mongoose
```

```
var passportLocalMongoose = require("passport-local-mongoose")
```

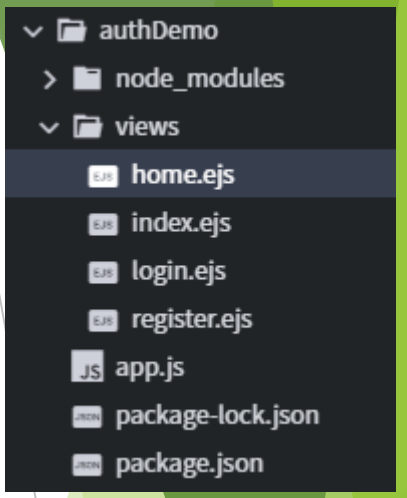
```
var schema = new mongoose.Schema({

    username: String,
    password: String

})

// Ads passportLocalMongoose methods to the schema
schema.plugin(passportLocalMongoose)

var User = mongoose.model("User", schema);
```

```
passport.use(new LocalStrategy(User.authenticate()))
passport.serializeUser(User.serializeUser())
passport.deserializeUser(User.deserializeUser())
```



Auth Demo Project

Auth Demo Project



- Create new Demo Project to test authorization
- Init npm
- Install required packages
- All required packages can be install in one go.

```
root@goorm:/workspace/MDS_Class/authDemo# npm install express express-session body-parser passport passport-local passport-local-mongoose mongoose ejs --save
```

- Create app.js file
- In the views directory create home, index, login, register ejs files
- Run ./mongod on an other terminal

App.js file

- Require the required packages and save them to respective variables.
- We require express, mongoose and body-parser.
- Additionally we also require passport which will be used to check if the user is authenticated.
- Local Strategy will be used as we will be using passport-local i.e user and password for authentication.
- We also be using passport –local-mongoose for storing user login and password information is the User model.
- We also set mongoose variables.
- Connect mongoose to a local database to store the user login and password database

```
var express = require("express")
var mongoose = require("mongoose")
var bodyParser = require("body-parser")

var passport = require("passport")
var LocalStrategy = require("passport-local")

var passportLocalMongoose = require("passport-local-mongoose")

// just to remove depracation warnings
mongoose.set('useNewUrlParser', true);
mongoose.set('useFindAndModify', false);
mongoose.set('useCreateIndex', true);
mongoose.set('useUnifiedTopology', true);

mongoose.connect("mongodb://localhost/authdemo_db")
```

App.js - Use Express Session/ Init passport

- Next we create the express app.
- We set the apps to use express-session
- Express-Session we need to set a secret string, and set resave and saveUninitiallised both to false.
- Then we set the app to use passport and session.
- We also set the app to use bodyParser as well.

```
var app = express()

app.use(require("express-session")({

    // will be used to encode and decode the sessions
    secret: "this is a secret phrase only I know",

    // just required
    resave: false,
    saveUninitialized: false

}))

// will initialize passport and
app.use(passport.initialize())
app.use(passport.session())


app.use(bodyParser.urlencoded({extended: true}));
```

App.js – User schema and model

- We create a mongoose schema which stores username and password.
- Use the plugin property for the schema and pass in the passportLocalMongoose var into it so that it adds the required methods to the schema.
- Create a model called User by passing “User” string and the schema to the mongoose.model function.
- The User var will be used when we create a new user.

```
// ++++++++++++++++++++++++++++++++++++++++
// MODELS
// ++++++++++++++++++++++++++++++++++++++++

var schema = new mongoose.Schema({

    username: String,
    password: String

})

// Ads passportLocalMongoose methods to the schema
schema.plugin(passportLocalMongoose)

var User = mongoose.model("User", schema);
```

App.js – Set Passport properties

- Finally we can set the strategy to passport.
- And pass in serialize and reserialize functions used passport to serialize and deserialize the User.
- This reads the session and takes the data and encodes and decodes the User data passed in.
- The serializeUser and deSeralizeUser functions were added into the User schema when set the passport- local- mongoose plugin into the User variable.

```
passport.use(new LocalStrategy(User.authenticate()))
passport.serializeUser(User.serializeUser())
passport.deserializeUser(User.deserializeUser())
```

Home.ejs

- The home.ejs file will be shown by going to the root route **GET** request of the server.
- It just has a header and 3 hrefs to signup, login and logout.

```
<h1>THIS IS THE HOME PAGE</h1>

<li><a href="/register">SignUp!</a></li>
<li><a href="/login">Login!</a></li>
<li><a href="/logout">Logout!</a></li>
```

```
app.get("/", function(req, res){

    res.render("home.ejs")

})
```

Register – GET/ POST

- Register **GET** route will show the register.ejs file which has a form for registering a user.
- Form sends the information to /register route **POST** request.

- The POST route, we can get user login and password data using body parser.
- Using the register function of user we can create a new user by passing in the user name, password and a callback function.
- If there is an error the register.ejs file is shown again.
- If there is no error we authenticate and serialize the user data using the local strategy and Redirect to the secret route GET request.

```
<h1>SIGN-UP FORM</h1>

<form action"/register" method= "POST">


<input type="text" name="username" placeholder"username">
<input type="password" name="password" placeholder"password">
<button>SUBMIT</button>

</form>
```

```
// show sign-up form
app.get("/register", function(req,res){


    res.render("register.ejs")
})
```

```
//handle user sign up
app.post("/register", function(req,res){

    var username = req.body.username
    var password = req.body.password

    // we dont save the password, it is encodes the password
    // and will return the user.

    User.register(new User({username: username}), password, function(err, user){

        if(err){
            console.log(err)
            return res.render("register.ejs")
        }else{
            // log the user in, run the serializeUser method
            // using the local strategy

            passport.authenticate("local")(req, res, function(){
                res.redirect("secret")

            })
        }
    })
})
```

Login – GET/ POST

```
<h1>LOGIN FORM</h1>

<form action"/login" method= "POST">

    <input type="text" name="username" placeholder"username">
    <input type="password" name="password" placeholder"password">
    <button>LOGIN</button>

</form>


<li><a href="/register">SignUp!</a></li>
<li><a href="/login">Login!</a></li>
<li><a href="/logout">Logout!</a></li>
```

- And login **GET** route shows login.ejs file for the user to logging in.
- The form sends the information to /login **POST** request.
- In the Login POST request,
- We pass in the POST route,
- In the second parameter we pass in passport.authenticate() function, which takes in a local strategy and takes in an object which tells the application which **GET** request route to go to on success or failure of authentication.
- The third parameter of the POST function is a callback which can be implemented if required.

```
// Show Login Form
app.get("/login", function(req,res){

    res.render("login.ejs")
})
```

```
app.post("/login", passport.authenticate("local",{

                // middleware runs before the callback
                // function
                successRedirect: "/secret",
                failureRedirect: "/login"

            }),function(req,res){ // callback

})
```

Secret Get request/ Index.ejs

- The index.ejs file will be shown by going to the secret route **GET** request of the server.
- **This page should only be shown when user is registered or logged In.**
- The request takes in a function **isLoggedIn**, this is called a **middleware**, which checks if a user is logged in.
- If user is Authenticated, then res.render() will be called else user will be redirected to login GET route.
- As you are authorized now, the get route shows a header saying that it is the secret page.
- Index.ejs has options for signup, login and **logout**.

```
<h1>THIS IS A SECRET PAGE</h1>

<p>YOU SHOULD ONLY BE ABLE TO SEE THIS PAGE IF YOU ARE LOGGED IN !!!</p>

<li><a href="/register">SignUp!</a></li>
<li><a href="/login">Login!</a></li>
<li><a href="/logout">Logout!</a></li>
```

```
app.get("/secret", isLoggedIn, function(req, res){

    res.render("index.ejs")
})
```

```
function isLoggedIn(req, res, next){

    if(req.isAuthenticated()){

      return next();
    }

    res.redirect("/login")

}
```

Logout GET request

- The logout route logs out the user by calling the logout() function of the request.
- Then redirect the user back to the root route.

```
app.get("/logout", function(req, res){

    req.logout();
    res.redirect("/")


})


app.listen(process.env.port || 3000, process.env.IP, function(){

    console.log("SERVER STARTED");

})
```

- Finally make sure to add the app.listen() function which actually starts the server.
- With **mongod** running, run node app.js now you will be able to see the secret page only after registering/ loging in into the web app.
- You can also check in the **mongo shell**, if the database is storing your user login and password data in the database.
- Notice the password is stored as a hash in the database.

```
> ls
[native code]
> show dbs
admin        0.000GB
authdemo_db  0.000GB
cars_db      0.000GB
config       0.000GB
db_app       0.000GB
local        0.000GB
> use authdemo_db
switched to db authdemo_db
> show collections
users
> db.users.find()
{ "_id" : ObjectId("5ec05c8ea636a409451338ac"), "username" : "siddharth.shekar@gmail.com", "salt" : "
8ede824650cbf7b20ff9c3b1e974909e46a0e201103bcfaabacc579ed7fb3398cdcb84c358a0221d88b4e602b047003a49a16
bfd8df7e411bfd6a79448864948b28c156f546065e61a68a3d39f77896f9d93aaf682366ba4a960706bd86713cccb1fb0732e
609bdffde47d6b8751cae47f01ff6b84fe0446e19b5c83f0e5e262898b86aabebec23cd162be8fad6ff6f0ba857f8be5f5490
52a2492d61d13e9a3911a225272f3eef86efbd59e4d339d877839869de9f8d04442930114c95c3b1535a56187a6a940e573de
ddc55567253d6fe0eb0d14499fa7d3cac94637e5297426d483e4090eedc1092d628749a0a674b0dfef5740605dea2ed4b5b37
{ "_id" : ObjectId("5ec05cd3a636a409451338ad"), "username" : "sidshekar", "salt" : "6cdc6b6cde0c6c142
6a54c289cf16f6d9c07816b1e70bc9e6e915f3e945b3b8f31933277e3229ad56bfa29957bea7ba8aa7eaccb200ad747a4283b
cfcc42aa9cadf8d528da9a658c7415928336dc92ecb63b1c94cb2b994c12b050583e1b629e4918f525593f518dab8224702
c8a970a4784f23ad9a6519c95546d270601c8f8265e0caea21d01454a9729d3ebd29116ee770f8efedc2764b9549cf16aa552
199ef7698ac7f47b39460b3d387fc3cd46ba17fced0ff818a13397ca68a8742ecc2442441009ee1ec644787f9aa8aa649351d
81c2a807b288d8010a1d96994b79f4294b208c52ddd8d3b0b45cdb2b475f4bead8a73c4933668c8adc807c5cff7ceb0c764d1
> 
```

Implementing Auth in ECarShop Project

- Now we will see how to link up and post with an authenticated user.
- In the EcarShop project
- Install passport, passport-local, passport-local-mongoose and express-session.
- Create 2 files called login.ejs and register.ejs in the views directory.

```
root@goorm:/workspace/MDS_Class/ECarShop/v2# ls
app.js  node_modules  package-lock.json  package.json  views
root@goorm:/workspace/MDS_Class/ECarShop/v2# touch views/login.ejs
root@goorm:/workspace/MDS_Class/ECarShop/v2# touch views/register.ejs
root@goorm:/workspace/MDS_Class/ECarShop/v2# npm install passport passport-local passport-local-mongoose express-session --save
npm WARN ecarshop@1.0.0 No description
npm WARN ecarshop@1.0.0 No repository field.

+ passport-local@1.0.0
+ passport@0.4.1
+ passport-local-mongoose@6.0.1
+ express-session@1.17.1
added 14 packages from 9 contributors and audited 106 packages in 6.915s
found 0 vulnerabilities
```

App.js – Requires and Car model

- Require the packages installed for passport, passport-local and passport-local-mongoose.

```
var express = require("express");
var bodyParser = require("body-parser");
var mongoose = require("mongoose");

var passport = require("passport")
var LocalStategy = require("passport-local")
var passportLocalMongoose = require("passport-local-mongoose")

var app = express();
app.use(bodyParser.urlencoded({extended: true}));

// just to remove depracation warnings
mongoose.set('useNewUrlParser', true);
mongoose.set('useFindAndModify', false);
mongoose.set('useCreateIndex', true);
mongoose.set('useUnifiedTopology', true);

mongoose.connect("mongodb://localhost/cars_db");
```

- Make changes to the Car model.
- So that it also takes a new obj called **author** which saves the password as an Id and also stores the username of the user as a string.

```
var carSchema = new mongoose.Schema({

    name: String,
    image: String,

    author:{
        id: {
                type: mongoose.Schema.Types.ObjectId,
                ref:"User"
            },

        username: String,
    },

});
```

App.js – User model/ Passport Config

- Create a new Model called User which stores the user's name and password as strings in the Schema.
- Set the schema to use the passport-local-mongoose as a plugin.

- Set passport to use local strategy for the user and set it to serialize and desacralize the user data.
- Set app to use express-session.
- Initialize passport().
- And set the app to use passport session().

```
var schema = new mongoose.Schema({

    username: String,
    password: String,

})

schema.plugin(passportLocalMongoose);

var User = mongoose.model("User", schema)
```

```
// ++++++++++++++++++++++++++++++++++++++++++++++++++
// PASSPORT CONFIGURATION
// ++++++++++++++++++++++++++++++++++++++++++++++++++

passport.use(new LocalStategy(User.authenticate()))
passport.serializeUser(User.serializeUser())
passport.deserializeUser(User.deserializeUser())


app.use(require("express-session")({
    secret:"car user author database",
    resave: false,
    saveUninitialized: false,

    }))


app.use(passport.initialize())
app.use(passport.session())
```

App.js – Register GET/ POST

- We add in the Register GET route which just displays the register.ejs file.
- Which has a form that takes username and password and sends it to the Register POST route.
- And the POST route creates a new user.
- Registers the user
- Once user is authenticated sends the user to the /cars route

```
// ++++ SIGN-UP ++++
app.get("/register", function(req, res){

    res.render("register.ejs")
})
```

```
app.post("/register", function(req,res){

    var username = req.body.username
    var password = req.body.password

    var newUser = new User({username: username});

    User.register(newUser, password, function(err, user){

        if(err){
            console.log(err);
            return res.render("register.ejs");
        }

        passport.authenticate("local")(req,res, function(){

            res.redirect("/cars/")

        });
    })
})
```

App.js - Login GET/ POST & Logout

- Login GET shows the login.ejs file which takes the login and password of user in a form and send the information to the Login POST route.
- Login POST route authenticates the user and redirects to the /cars route if the user is authenticated else sends to /login route if it fails.
- Add the Logout Route as well.

```
//++++ LOGIN ++++
app.get("/login",  function(req, res){

    res.render("login.ejs")

})


app.post("/login", passport.authenticate("local",{

                    successRedirect: "/cars",
                    failureRedirect: "/login"

                }),function(req,res){ // callback

})
```

```
// ++++ LOGOUT ++++
app.get("/logout", function(req, res){

    req.logout();
    res.redirect("/cars")


})
```

Header.ejs and index.ejs

- Make changes in the header.ejs file.
- So that when the login/ register and logout buttons are pressed on the Navbar the href is pointing to the correct login and register routes.

- Make changes to the index.ejs file so that you are also displaying the user name who submitted the post.

```
<img src="<%= car.image %>">

<div class="caption">
    <h4>    <%= car.name %>  </h4>
</div>
<p>
    <em>Submitted By: <%= car.author.username %></em>
</p>
```

App.js – Cars POST Route and /cars/new Route.

- We change the POST request to check if the user is logged in by using the same isLoggedIn function from AuthDemo project.
- We make changes to how newCar is created, so that when we add a new car we store the user information apart from car name and image.
- Then create a new Car.
- Save the information to the DB
- Then redirect to the /Cars route.

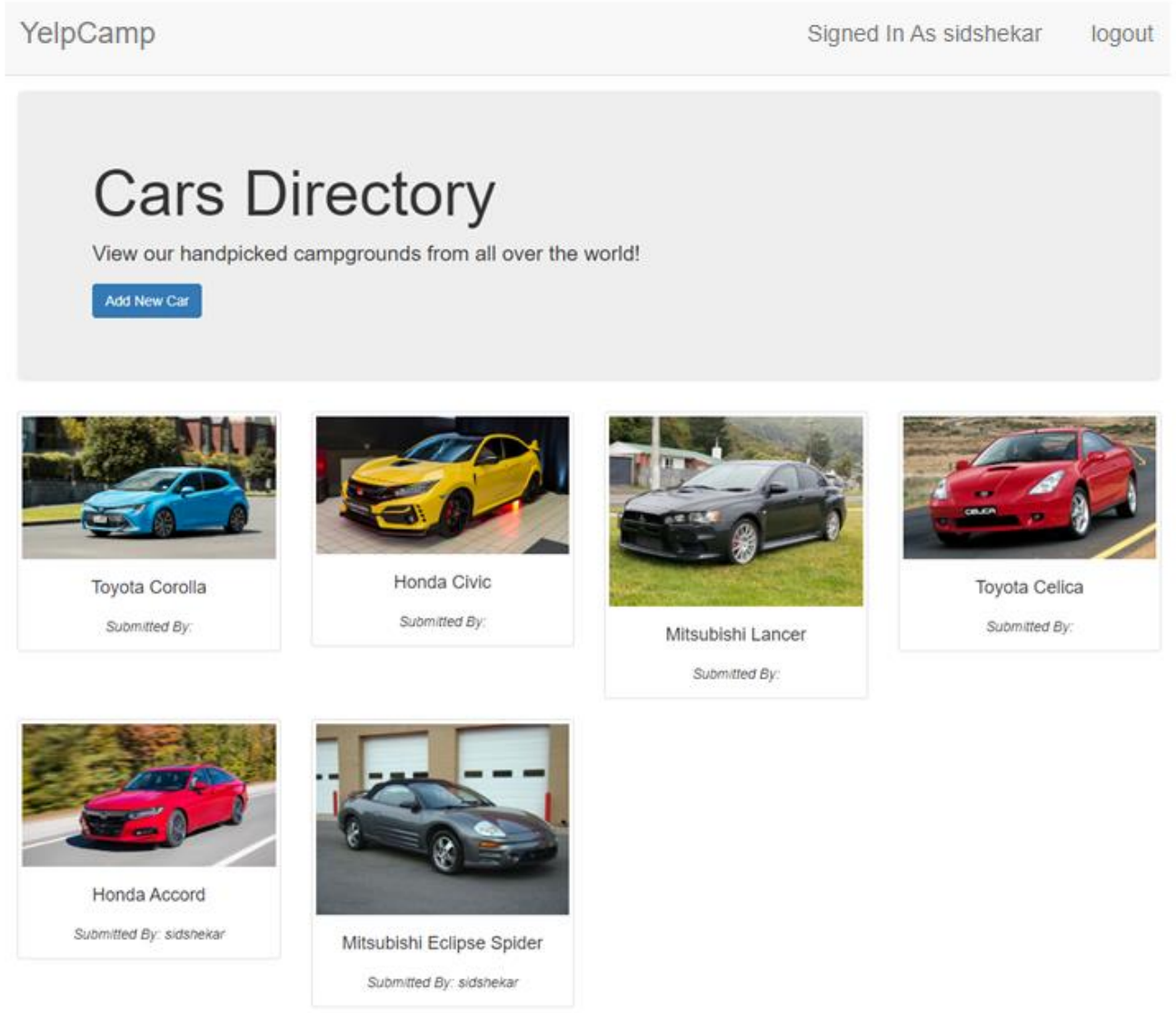
- In the car/new route also we check if the user is logged in to show the add new car ejs file.

```
//CREATE - add new campground
app.post("/cars", isLoggedIn, function(req, res){

    // get data from db and add to cars array
    var name= req.body.name;
    var image = req.body.image;

        var author = {
        id: req.user._id,
        username: req.user.username
    }

    var newCar = {name: name, image: image, author: author};

    var car = new Car(newCar);
    car.save(errCallBack);


    // redirect back to cars page
    res.redirect("/cars"); //default of redirecting is a GET request

})
```

```
// NEW - Show form to create new car
app.get("/cars/new", isLoggedIn, function(req,res){

    res.render("new.ejs");
})
```

Required Output

- Now when you click Add New Car in the index.ejs page, if you are not logged in, you will be taken to the login page.
- Login to add a post, or if you are not signed up, sign up first.
- Once signed up you will now be able to add a new car and image.
- Then you will be redirected to the index.ejs file,
- In it, for the new cars added, along with the car name and image, the user name of the user who submitted the information will also be displayed.



Exercise

- To show the login/register button in the navbar if user is not logged in or registered.
- And Once user is logged in show only Signedin As user and Logout button.
- Create a new middlware in app.js file which takes current user data and stores the data in res.locals.currentUser.
- In the **/cars GET route** set the req.user to the user variable.
- Now currentUser variable can be accessed in the header.ejs where the Navbar implementation is present.

```
//our middlware
app.use(function(req,res, next){

    res.locals.currentUser = req.user;
    next();
})
```

```
var user = req.user;
```

```
<ul class="nav navbar-nav navbar-right">
    <% if(!currentUser){ %>
    <li><a href="/login">login</a></li>
    <li><a href="/register">SignUp</a></li>
    <% }else{ %>
    <Li><a href="#">Signed In As <%= currentUser.username%></a></Li>
    <li><a href="/logout">logout</a></li>
    <% } %>
</ul>
```