AIoTDesign Platform Design

Lessons I learned from this instruction:

1. The order of `app.use` statements is very important.

Step 1. Installing Node

You will first install *Nodejs* and the Node Package Manager (NPM) on your operating system. The following sections explain the easiest way to install the Long Term Supported (LTS) version of Nodejs on Windows 10.

1. Go to https://nodejs.org/en/
2. Select the button to download the LTS build that is "Recommended for most users".
2. Install Node by double-clicking on the downloaded file and following the installation prompts.

Step 2. Installing Express

Express is a lightweight web application framework for Node.js, which provides us with a robust set of features for writing web apps. These features include such things as route handling, template engine integration and a middleware framework, which allows us to perform additional tasks on request and response objects. There's nothing you can do in Express that you couldn't do in plain Node.js, but using Express means we don't have to re-invent the wheel and it reduces boilerplate. Now install Express:

```
npm install express
```

Step 3. Install nodemon

nodemon is a convenience tool. It will watch the files in the directory it was started in, and if it detects any changes, it will automatically restart your Node application (meaning you don't have to). In contrast to Express, nodemon is not something the app requires to function properly (it just aids us with development), so install it using:

```
npm install --save-dev nodemon
```

This will add nodemon to the `dev-dependencies` section of the `package.json` file.

Step 4. Installing Express Application Generator

The Express Application Generator tool generates an Express application "skeleton". Install the generator using NPM as shown (the -g flag installs the tool globally so that you can call it from anywhere):

```
C:\Users\fwang>npm install express-generator -g

npm WARN deprecated mkdirp@0.5.1: Legacy versions of mkdirp are no longer supported. Please update to mkdirp 1.x. (Note that the API surface has changed to use Promises in 1.x.)

C:\Users\fwang\AppData\Roaming\npm\express -> C:\Users\fwang\AppData\Roaming\npm\node_modules\express-generator\bin\express-cli.js

+ express-generator@4.16.1

added 10 packages from 13 contributors in 1.367s
```

Step 5. Create a project folder

```

C:\Users\fwang>mkdir AIoTDesign
```

Step 6. Create a project skeleton

Navigate to the project and then run the Express Application Generator in the command prompt as shown:

```
C:\Users\fwang>cd AIoTDesign
```

```
C:\Users\fwang\AIoTDesign>express aiotdesign --view=hbs

   create : aiotdesign\
   create : aiotdesign\public\
   create : aiotdesign\public\javascripts\
   create : aiotdesign\public\images\
   create : aiotdesign\public\stylesheets\
   create : aiotdesign\public\stylesheets\style.css
   create : aiotdesign\routes\
   create : aiotdesign\routes\index.js
   create : aiotdesign\routes\users.js
   create : aiotdesign\views\
   create : aiotdesign\views\error.hbs
   create : aiotdesign\views\index.hbs
   create : aiotdesign\views\layout.hbs
   create : aiotdesign\app.js
   create : aiotdesign\package.json
   create : aiotdesign\bin\
   create : aiotdesign\bin\www

   change directory:
     > cd aiotdesign

   install dependencies:
     > npm install

   run the app:
     > SET DEBUG=aiotdesign:* & npm start
```

Step 7. Modify dependencies

The dependencies include the express package and the package for our selected view engine. Now we're going to go cd aiotdesign:

```
$ cd aiotdesign
```

let's open up package.json and you can see it's included a bunch of dependencies here:

```
{
  "name": "aiotdesign",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "*",
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "express": "~4.16.1",
    "express-handlebars": "*",
    "http-errors": "~1.6.3",
    "morgan": "~1.9.1",
    "serve-favicon": "~2.3.0",
    "bcryptjs":"*",
    "passport":"*",
    "passport-http":"*",
    "passport-local":"*",
    "mongodb":"*",
    "mongoose":"*",
    "express-session":"*",
    "connect-flash":"*",
    "express-messages":"*",
    "express-validator":"*"
  }
}
```

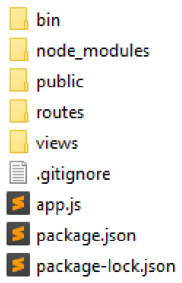
Step 8. Running the skeleton project

At this point, we have a complete skeleton project. The website doesn't actually *do* very much yet, but it's worth running it to show that it works.

1. First, install the dependencies (the `install` command will fetch all the dependency packages listed in the project's **package.json** file).

```
npm install
```

Here is the final tree structure after installing dependencies.



```
bin
node_modules
public
routes
views
.gitignore
app.js
package.json
package-lock.json
```

2. Then run the application.

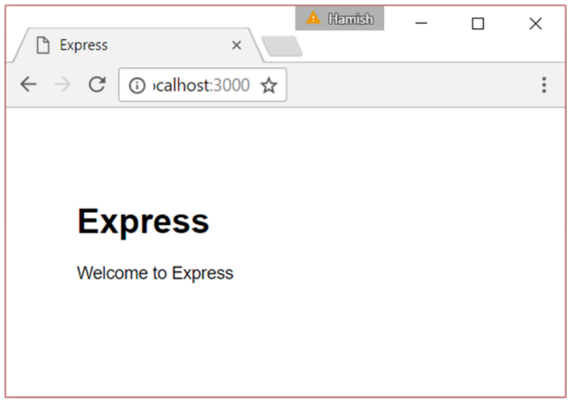
   On Windows, use this command:

```
SET DEBUG=aiotdesign:* & npm start
```

```
C:\Users\fwang\AIoTDesign\aiotdesign>SET DEBUG=aiotdesign:* & npm start

> aiotdesign@0.0.0 start C:\Users\fwang\AIoTDesign\aiotdesign
> node ./bin/www

  aiotdesign:server Listening on port 3000 +0ms
GET / 200 99.298 ms - 204
GET /stylesheets/style.css 200 3.758 ms - 111
GET / 304 6.657 ms - -
GET /stylesheets/style.css 304 1.038 ms - -
```

3. Then load http://localhost:3000/ in your browser to access the app.

You should see a browser page that looks like this:



Express

Welcome to Express

Step 9. Modify the scripts section

The scripts section defines a "*start*" script, which is what we are invoking when we call `npm start` (by default) to start the server. From the script definition, you can see that this actually starts the JavaScript file **./bin/www** with *node*. It also defines a "*devstart*" and a “serverstart” scripts, which we invoke when calling `npm run serverstart` instead. This starts the same **./bin/www** file, but with *nodemon* rather than *node*.

```
…
  "express-messages": "*",
  "express-validator": "*"
},
"scripts": {
  "start": "node ./bin/www",
  "devstart": "nodemon ./bin/www",
  "serverstart": "SET DEBUG=aiotdesign:* & npm run devstart"
},
```

Then run the application:

```
C:\Users\fwang\AIoTDesign\aiotdesign>npm run serverstart

> aiotdesign@0.0.0 serverstart C:\Users\fwang\AIoTDesign\aiotdesign
> SET DEBUG=aiotdesign:* & npm run devstart



> aiotdesign@0.0.0 devstart C:\Users\fwang\AIoTDesign\aiotdesign
> nodemon ./bin/www

[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./bin/www`
  aiotdesign:server Listening on port 3000 +0ms
GET / 200 109.274 ms - 204
GET /stylesheets/style.css 200 4.031 ms - 111
GET /favicon.ico 404 10.768 ms - 1217
```

Step 10. Understand www file

The file **/bin/www** is the application entry point! The very first thing this does is `require()` the "real" application entry point (**app.js**, in the project root) that sets up and returns the `express()` application object.

```
#!/usr/bin/env node

/**
 * Module dependencies.
 */
var app = require('../app');
```

Note: `require()` is a global node function that is used to import modules into the current file. Here we specify **app.js** module using a relative path and omitting the optional (**.js**) file extension.

The remainder of the code in this file sets up a node HTTP server with `app` set to a specific port (defined in an environment variable or 3000 if the variable isn't defined), and starts listening and reporting server errors and connections. For now you don't really need to know anything else about the code (everything in this file is "boilerplate"), but feel free to review it if you're interested.

Step 11. Understand app.js

This file creates an `express` application object (named `app`, by convention), sets up the application with various settings and middleware, and then exports the app from the module. The code below shows just the parts of the file that create and export the app object:

```
var express = require('express');
var app = express();
...
module.exports = app;
```

Back in the **www** entry point file above, it is this `module.exports` object that is supplied to the caller when this file is imported.

Let's work through the **app.js** file in detail. First, we import some useful node libraries into the file using `require()`, including http-errors, *express*, *morgan* and *cookie-parser* that we previously downloaded for our application using NPM; and *path*, which is a core Node library for parsing file and directory paths.

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');
```

Then we `require()` modules from our routes directory. These modules/files contain code for handling particular sets of related "routes" (URL paths). When we extend the skeleton application, we will add a new file for dealing with book-related routes.

```
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
```

Note: At this point, we have just *imported* the module; we haven't actually used its routes yet (this happens just a little bit further down the file).

Next, we create the `app` object using our imported *express* module, and then use it to set up the view (template) engine. There are two parts to setting up the engine. First, we set the '`views`' value to specify the folder where the templates will be stored (in this case the subfolder **/views**). Then we set the '`view engine`' value to specify the template library (in this case "hbs").

```
var app = express();

// view engine setup
```

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');
```

The next set of functions call `app.use()` to add the *middleware* libraries into the request handling chain. In addition to the 3rd party libraries we imported previously, we use the `express.static` middleware to get *Express* to serve all the static files in the **/public** directory in the project root.

```
app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

Now that all the other middleware is set up, we add our (previously imported) route-handling code to the request handling chain. The imported code will define particular routes for the different *parts* of the site:

```
app.use('/', indexRouter);
app.use('/users', usersRouter);
```

Note: The paths specified above ('/' and '`/users`') are treated as a prefix to routes defined in the imported files. So for example, if the imported **users** module defines a route for `/profile`, you would access that route at `/users/profile`. We'll talk more about routes in a later article.

The last middleware in the file adds handler methods for errors and HTTP 404 responses.

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});
```

The Express application object (app) is now fully configured. The last step is to add it to the module exports (this is what allows it to be imported by **/bin/www**).

```
module.exports = app;
```

Step 12. Configuring the app.js file

Now what you will do is open app.js and there's a bunch of stuff that we have to include up there:

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');
var favicon = require('serve-favicon');
var bodyParser = require('body-parser');
var exphbs = require('express-handlebars');
var expressValidator = require('express-validator');
var flash = require('connect-flash');
var session = require('express-session');
var passport = require('passport');
var LocalStrategy = require('passport-local'),Strategy;
var mongo = require('mongodb');
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/elearn');
var db = mongoose.connection;
async = require('async');
```

Step 13. Modify view engine section

Now we change the view engine from "hbs" to "express-handlebars".

First, you will install `express-handlebars, handlebars and @handlebars/allow-prototype-access:`

```
C:\Users\fwang\AIoTDesign\aiotdesign>npm install express-handlebars

C:\Users\fwang\AIoTDesign\aiotdesign>npm install -D handlebars@4.7.2

C:\Users\fwang\AIoTDesign\aiotdesign>npm install @handlebars/allow-prototype-access
```

Starting from version 4.6.0, Handlebars forbids accessing prototype properties and methods of the context object by default. This is related to a security issue described here: https://mahmoudsec.blogspot.com/2019/04/handlebars-template-injection-and-rce.html. In the past, Handlebars would allow you to access prototype methods and properties of the input object from the template. Multiple security issues have come from this behaviour. If you are certain that only developers have access to the templates, it's possible to allow prototype access by installing the following package:

```
npm install @handlebars/allow-prototype-access
```

This package can help you disable prototype checks for your models. In handlebars@^4.6.0. access to the object prototype has been disabled completely. Now, if you use custom classes as input to Handlebars, your code won't work anymore. This package automatically adds runtime options to each template-calls, disabling the security restrictions. In `app.js` file, we add the `handlebars` package before "`var exphbs = require('express-handlebars')`" statement and a statement after that:

```
var Handlebars = require('handlebars');
var exphbs = require('express-handlebars');

// Import function exported by newly installed node modules.
const { allowInsecurePrototypeAccess } = require('@handlebars/allow-prototype-access');
```

Next, we modify the view engine in the `app.js` file. We're going to keep the first line of the two, because it's just telling us to use the folder called views. But for the second one, we're going to change "hbs" to handlebars and then there's one additional line that we need to include here, which is `app.engine`, and we're going to pass in handlebars. Then we want to take that variable that we created above exphbs and just pass in an object. We want to define the default layout. This will be the name of the file that you want to use for your layout. We're just going to call it layout. The view engine setup is going to look like this:

```
// view engine setup
app.set('views', path.join(__dirname, 'view'));
app.engine('handlebars', exphbs({
    defaultLayout: 'layout',
    // ...implement newly added insecure prototype access
```

```
        handlebars: allowInsecurePrototypeAccess(Handlebars)
        })
    );
    app.set('view engine', 'handlebars');
```
That's all we have to do to set up Express Handlebars.

Now there's some other middleware that we need to add. So we're going to the `app.use` statements, and the first thing we're going to add is for the Express session:

```
app.use(express.static(path.join(__dirname, 'public')));
// Express Session
app.use(session({
    secret: 'secret',
    saveUninitialized: true,
    resave: true
}));
```
It's just providing a secret. You can change this if you'd like. The next thing is going to be for our Passport. We're going to want to call initialize as well as session:

```
// Passport
app.use(passport.initialize());
app.use(passport.session());
```
The next thing is for Express Validator, and this is right from the GitHub page. I haven't changed anything:

```
// Express Validator
app.use(expressValidator({
  errorFormatter: function(param, msg, value) {
      var namespace = param.split('.')
      , root    = namespace.shift()
      , formParam = root;

    while(namespace.length) {
      formParam += '[' + namespace.shift() + ']';
    }
    return {
      param : formParam,
      msg   : msg,
      value : value
    };
  }
}));
```

Note that Express Validator has been updated so that you can can't use the above code in the latest version. However, you will use previous version by running this commands:

```
npm uninstall express-validator
npm install express-validator@5.3.0
```

Finally, we have our `connect-flash` and `express-messages`:

```
// Connect-Flash
app.use(flash());

// Global Vars
app.use(function(req, res, next) {
  res.locals.messages = require('express-messages')(req, res);
  next();
});
```

We want to add `app.use(flash())` and then in `Global vars`, we're going to set a global variable called messages if there's anything we want to put out there. That should do it. Let's save that.

Step 14. Configuring the views directory

We'll now go to our views file and we have three files there: `error.hbs, index.hbs`, and `layout.hbs`. We'll configure them using handlebars. First, we're going to rename the `error.hbs` file to `error.handlebars`. For the content, we can get that out and we're just going to paste in an h1 in the paragraph like this:

```
<h1>Error</h1>
<p>You are in the wrong place</p>
```

Now there's some other middleware that we need to add. So we're going to the app.use statements, and the first thing we're going to add is for the Express session:

Then for `index.hbs`, we're going to rename that as well to index.handlebars. For that, we're just going to put in an h1 and we'll just say Welcome:

```
<h1>Welcome</h1>
```

Then we delete the `layout.hbs` file and create a folder called `layouts` because that's where Express Handlebars is going to look. Next, we're going to create a file called `layout.handlebars` in that folder. In layout.handlebars, let's put our base HTML and we'll add the title as AIoTDesign. Then, in the body, we just want to add triple curly braces {{{}}} and then add body:

```
<!DOCTYPE html>
<html>
  <head>
    <title>AIoTDesign</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

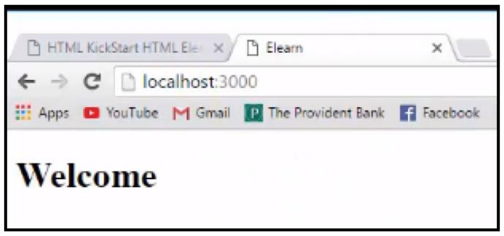
That will output whatever view we're at, at the current time. Let's save that.

Step 15. Running the setup in the browser

Now, let's test out our setup and run the npm start command:

$ npm run serverstart

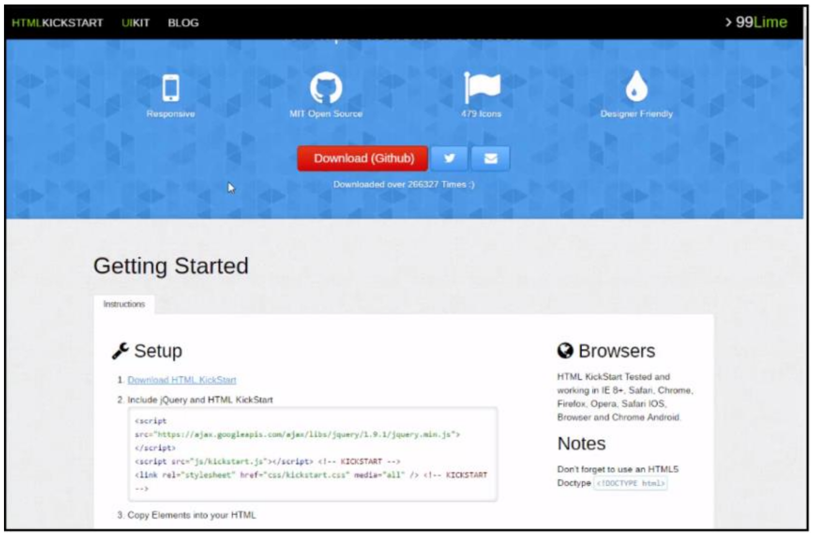
Then, we'll go to localhost:3000, and we get Welcome as shown here:



Welcome

Step 16. Implementing our layout

We're going to use HTML KickStart for implementing the layout:



Getting Started

Instructions

Setup

1. Download HTML KickStart
2. Include jQuery and HTML KickStart

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
</script>
<script src="js/kickstart.js"></script> <!-- KICKSTART -->
<link rel="stylesheet" href="css/kickstart.css" media="all" /> <!-- KICKSTART
-->
```

3. Copy Elements into your HTML

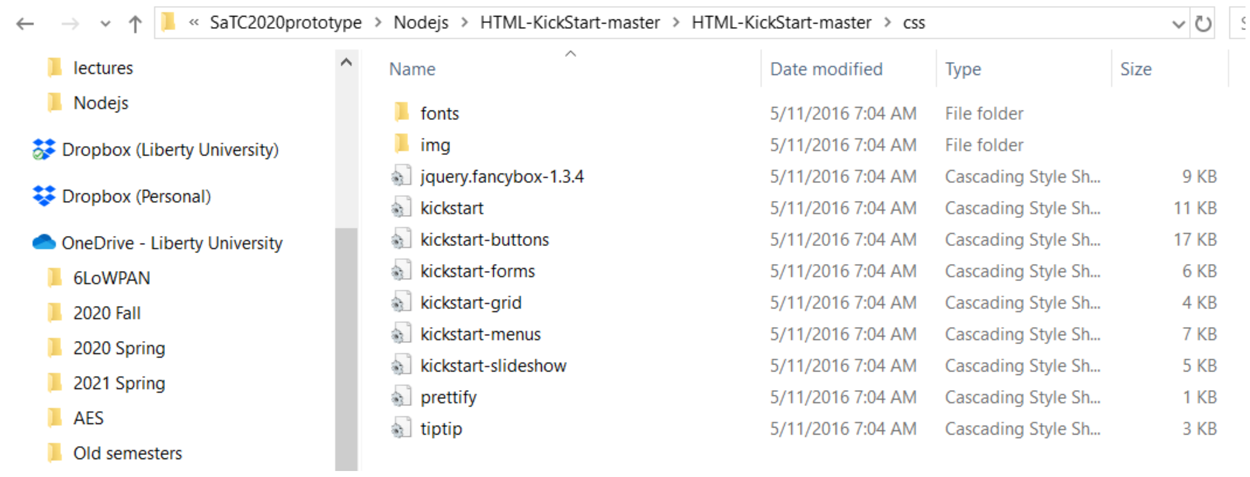
Browsers

HTML KickStart Tested and working in IE 8+, Safari, Chrome, Firefox, Opera, Safari IOS, Browser and Chrome Android.

Notes

Don't forget to use an HTML5 Doctype `<!DOCTYPE html>`

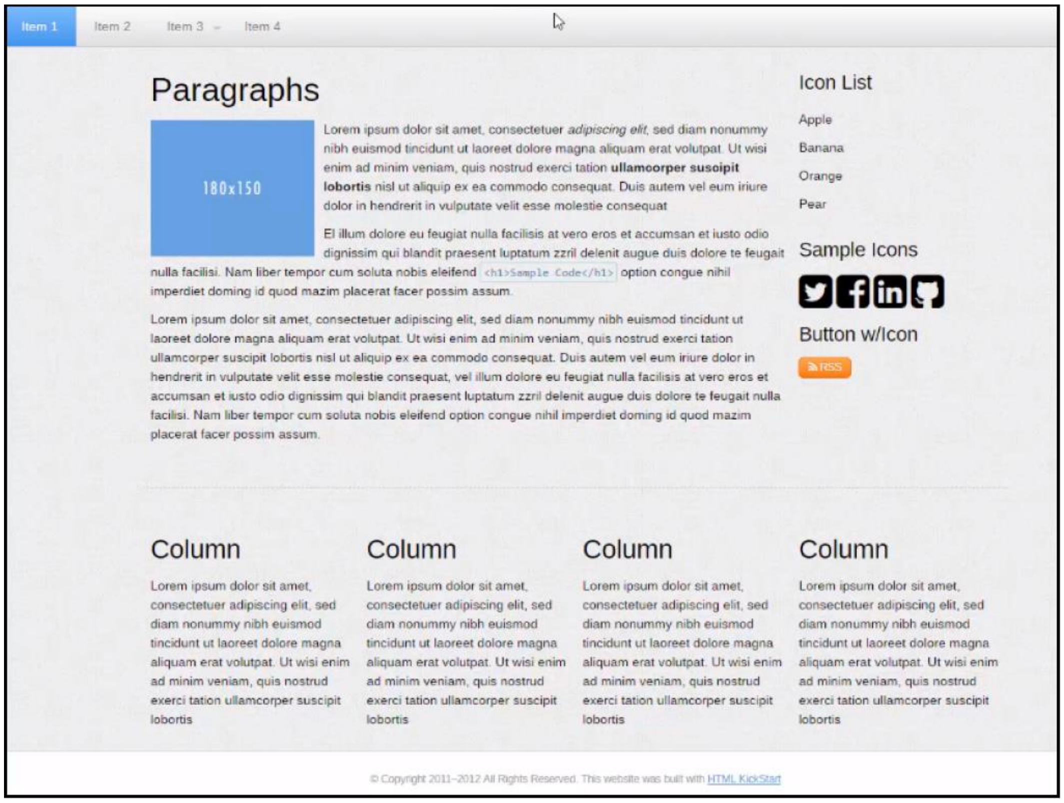
You need to click on the Download button shown in the preceding screenshot and that should start to download. We'll open that up and what we're going to do first is bring over the CSS stuff:


« SaTC2020prototype > Nodejs > HTML-KickStart-master > HTML-KickStart-master > css
lectures
Nodejs
Dropbox (Liberty University)
Dropbox (Personal)
OneDrive - Liberty University
6LoWPAN
2020 Fall
2020 Spring
2021 Spring
AES
Old semesters

Name	Date modified	Type	Size
fonts	5/11/2016 7:04 AM	File folder	
img	5/11/2016 7:04 AM	File folder	
jquery.fancybox-1.3.4	5/11/2016 7:04 AM	Cascading Style Sh...	9 KB
kickstart	5/11/2016 7:04 AM	Cascading Style Sh...	11 KB
kickstart-buttons	5/11/2016 7:04 AM	Cascading Style Sh...	17 KB
kickstart-forms	5/11/2016 7:04 AM	Cascading Style Sh...	6 KB
kickstart-grid	5/11/2016 7:04 AM	Cascading Style Sh...	4 KB
kickstart-menus	5/11/2016 7:04 AM	Cascading Style Sh...	7 KB
kickstart-slideshow	5/11/2016 7:04 AM	Cascading Style Sh...	5 KB
prettify	5/11/2016 7:04 AM	Cascading Style Sh...	1 KB
tiptip	5/11/2016 7:04 AM	Cascading Style Sh...	3 KB

I'm just going to copy everything that's shown in the preceding folder and paste it into the Projects | public | stylesheets folder. Then inside js, we're going to bring `kickstart.js` over to our JavaScripts folder..

Now let's open this example.html provided in the `HTML-kickstart-master` and this is what it looks like:



Paragraphs

Lorem ipsum dolor sit amet, consectetuer *adipiscing elit*, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation **ullamcorper susipit lobortis** nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat.

El illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend `<h1>Sample Code</h1>` option congue nihil imperdiet doming id quod mazim placerat facer possim assum.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum.

Icon List

Apple
Banana
Orange
Pear

Sample Icons

Button w/Icon

Column

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Column

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Column

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Column

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

© Copyright 2011–2012 All Rights Reserved. This website was built with HTML KickStart

What we'll do is we'll open that example.html with an editor. We'll grab the code that's in here, copy it, and bring it over to our layout.handlers. You will overwrite this next.

Step 17. Configuring the title and header in the layout

Starting at the top, let's take the title out and add AIoTDesign. Then, we don't need the meta stuff present in the code, so we'll remove this. Then, we'll move on to the link to the stylesheets. We know that we don't have a css folder; it's actually called stylesheets. You want to have / at the beginning of the stylesheets as well. We want it for the scripts as well. We want it for this one. That stylesheet was actually in the root of the framework, so we might have to bring that over. Then, we'll change the script type, javascript, to /javascripts like this:

```
<!DOCTYPE html>
<html><head>
<title>AIoTDesign</title>
<meta charset="UTF-8">
<link rel="stylesheet" type="text/css" href="/stylesheets/kickstart.css" media="all" />
<link rel="stylesheet" type="text/css" href="/stylesheets/style.css" media="all" />
<script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<script type="text/javascript" src="/javascripts/kickstart.js"></script>
</head><body>
```

Step 18 Configuring the body in the layout

This menu here is going to get much, much simpler. I will get rid of all the <li> below the first row <li>s:

```
<body>
<!-- Menu Horizontal -->
<ul class="menu">
<li class="current"><a href="">Item 1</a></li>
<li><a href="">Item 2</a></li>
</ul>
```

Then we're going to edit the Item 1 to Home. In the href, we'll add /, item 2 will be Classes, and the href will go to /classes:

```
<body>
<!-- Menu Horizontal -->
<ul class="menu">
<li class="current"><a href="/">Home</a></li>
<li><a href="/classes">Classes</a></li>
</ul>
Next to it we have our grid, we'll keep it as it is.
```

Step 19. Configuring the paragraph

In the paragraph, you will get rid of the image. In this h3, we'll just add Welcome To AIoTDesign. Then, we'll leave the content just so it's not completely empty, but we only keep the top paragraph:

```
<div class="col_12">
<div class="col_9">
<h3>Welcome To AIoTDesign</h3>
<p>
Lorem ipsum dolor sit amet, consectetuer, <em>adipiscing elit</em>,
sed diam
nonummy nibh euismod
tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim
ad
minim veniam, quis
nostrud exerci tation <strong>ullamcorper suscipit lobortis</strong>
nisl ut
aliquip ex ea commodo consequat.
Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse
molestie consequat
</p>
```

Step 20. Configuring the sidebar

Then for the sidebar, let's change the heading. You add Student &amp; Instructor Login. We're going to get rid of the ul and let's put a form. In the form, we're going to have a label. So this label will be Username and then we'll have an input with a type of text. Then we'll have another label Password. Let's give these inputs a name as well. Let's put a Submit button and give this a class="button":

```
 <div class="col_3">
   <h5>Student & Instructor Login</h5>
   <form>
   <label>Username: </label>
   <input type="text" name="username">
   <label>Password: <label>
   <input type="password" name="password">
   <input class="button" type="submit" value="submit">
   </form>
   </div>
```

We can then get rid of the stuff in the h5.

Step 21. Configuring hr

Now, under hr, you don't want four columns. We want three of them, and they're going to be classes. Let's get rid of this last one and then change the classes to a 4-column one. Then, we'll have our classes. Let's say HTML 101, Intro To PHP, and then let's say Learn Node.js:

```
            <hr />

            <div class="col_3">
            <h4>HTML 101</h4>
```

```
		<p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut
laoreet dolore
		magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit
lobortis</p>
		</div>

		<div class="col_3">
		<h4>Intro To PHP</h4>
		<p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut
laoreet dolore
		magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit
lobortis</p>
		</div>

		<div class="col_3">
		<h4>Learn Node.js</h4>
		<p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut
laoreet dolore
		magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit
lobortis</p>
		</div>

		<div class="col_3">
		<h4>Master Flask</h4>
		<p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut
laoreet dolore
		magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit
lobortis</p>
		</div>
</div>
```

Step 22. Configuring the footer in the layout

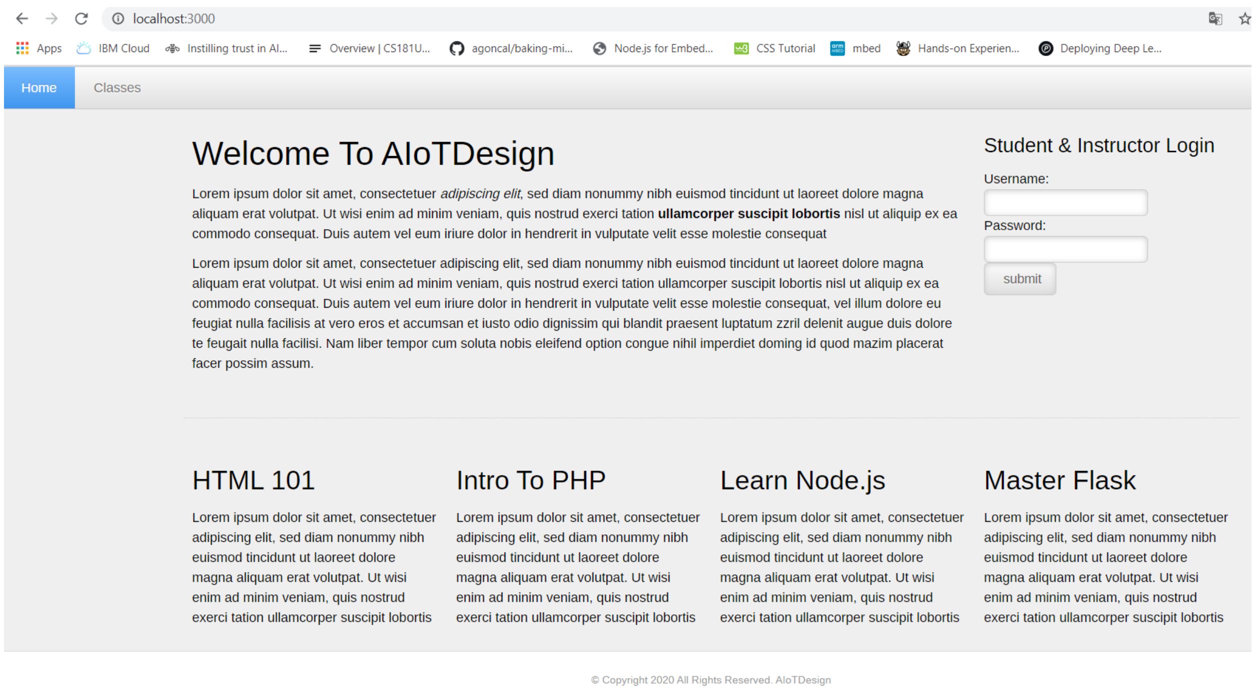
In the footer, we will change the HTML Kickstart to AIoTDesign and we'll add Copyright 2016 like this:

```
<!-- ================================ START FOOTER================================= -->
<div class="clear"></div>
<div id="footer">
&copy; Copyright 2018 All Rights Reserved. Elearn</a>
</div>
```

Let's save that and go to our app.

The final application

If we go to the app, this is how it is going to look:


localhost:3000

Home | Classes

Welcome To AIoTDesign

Lorem ipsum dolor sit amet, consectetuer *adipiscing elit*, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation **ullamcorper suscipit lobortis** nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum.

Student & Instructor Login

Username:

Password:

submit

HTML 101

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Intro To PHP

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Learn Node.js

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Master Flask

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

© Copyright 2020 All Rights Reserved. AIoTDesign

Now you'll notice that this looks a little weird considering the padding and background colors, and all that grey. The reason for that is because if we look at the HTML template that we have, there's a style.css file that's outside of the css folder for some reason. So what we're going to do is just copy that and then bring that over to our Projects | public |stylesheets folder and then we'll paste it there. There might be already a style.css file in this folder. In that case, we'll replace that file. Now, if we go back to our app, it looks a lot better.

Next, you will add some buttons to the classes in <hr>. So if we go in each of these columns in the HTML file, we're going to put a link with a class of button, and for now, we'll just set this to go nowhere and then that'll just say View Class:

```
<div class="col_4">
<h4>HTML 101</h4>
<p>Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam
nonummy
nibh euismod tincidunt ut laoreet dolore
magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud
exerci
tation ullamcorper suscipit lobortis</p>
<a class="button" href="#">View Class</a>
</div>
<div class="col_4">
<h4>Intro To PHP</h4>
<p>Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam
nonummy
nibh euismod tincidunt ut laoreet dolore
magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud
exerci
tation ullamcorper suscipit lobortis</p>
<a class="button" href="#">View Class</a>
</div>
<div class="col_4">
<h4>Learn Node.js</h4>
<p>Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam
nonummy
nibh euismod tincidunt ut laoreet dolore
magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud
exerci
tation ullamcorper suscipit lobortis</p>
```

```
<a class="button" href="#">View Class</a>
</div>
```

Now, we can also put the paragraph part into the index template. Let's grab everything that's in the col_9 div, cut that, and then we're going to put our {{{}}} tag thing in there:

```
<div class="col_12">
<div class="col_9">
{{{body}}}
</div>
```

Then let's go to index and we'll paste this in:

```
<h3>Welcome To Elearn</h3>
<p>
Lorem ipsum dolor sit amet, consectetuer, <em>adipiscing elit</em>, sed
diam
nonummy nibh euismod
tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad
minim
veniam, quis
nostrud exerci tation <strong>ullamcorper suscipit lobortis</strong>
nisl ut
aliquip ex ea commodo consequat.
Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse
molestie
consequat</p>
```

Step 23. Fetching classes – part A

In the last section, we went ahead and set up our application and we integrated Kickstart on the frontend. So we have a basic layout. What we want to do now is start to focus on getting classes from the database, and we want to fix the homepage so that the Login part is coming from its own file as well as the classes. These are going to be what's called partials.

1. Setting up partials

   We're going to create a folder inside the views folder, called views/partials. For now, we're going to have two files in here. We'll create one and let's save it as classes.handlebars. We also want one called login.handlebars.

   Let's start with the login. What we'll do is go to our layout and we're going to grab from the heading down to where the form ends, cut that out, paste it in the login.handlebars:

   ```
   <h5>Student & Instructor Login</h5>
   <form>
   <label>Username: </label>
   <input type="text" name="username">
   <label>Password: </label>
   <input type="password" name="password">
   <input class="button" type="submit" value="submit">
   </form>
   ```

   Then in place of that in layout.handlebars, we're going to use a {{>login}}:

   ```
   <div class="col_3">
   {{>login}}
   <div>
   ```

   That's how you can include a partial. So let's go ahead and save that. Then, we'll restart our app, and it should look like the previous one.

   Now we want to do the same for classes. So we're going to go back to our layout and grab all of the col_4 divs, cut that, and then in place of it we're going to say classes and we'll save that:

   ```
   <hr />
   {{>classes}}
   </div>
   ```

Now let's go into classes.handlebars, paste the column classes, and I'm also going to put a heading here, so we'll put an h4 and we'll say Latest Classes:

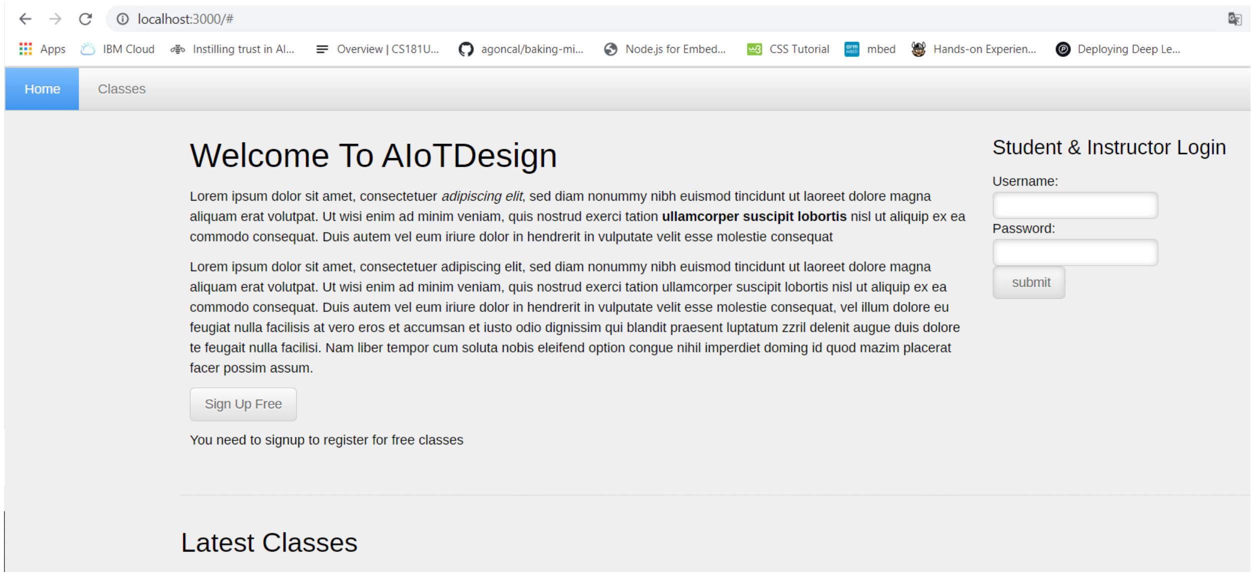
```
<h4>Latest Classes</h4>
<div class="col_4">
<h4>HTML 101</h4>
<p>Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam
nonummy
nibh euismod tincidunt ut laoreet dolore
magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
nostrud
exerci tation ullamcorper suscipit lobortis</p>
<a class="button" href="#">View Class</a>
</div>
<div class="col_4">
<h4>Intro To PHP</h4>
<p>Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam
nonummy
nibh euismod tincidunt ut laoreet dolore
magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
nostrud
exerci tation ullamcorper suscipit lobortis</p>
<a class="button" href="#">View Class</a>
</div>
<div class="col_4">
<h4>Learn Node.js</h4>
<p>Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam
nonummy
nibh euismod tincidunt ut laoreet dolore
magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
nostrud
exerci tation ullamcorper suscipit lobortis</p>
<a class="button" href="#">View Class</a>
</div>
```

Let's save that and now we have our classes in a partial.

Now we want to change the Welcome To AIoTDesign area a little bit and put some content in there and also a button to signup. Let's go to index.handlebars and paste some relevant content. So we just have a simple paragraph with some real text and then we have a Sign Up Free button, and then just a paragraph letting them know they need to sign up to register for classes:

```
<h3>Welcome To eLearn</h3>
<p>
AIoTDesign is a simple and free online learning platform. Instructors can create courses and students can
register for courses which include assignments, quizzes and support forums.
</p>
<a class="button" href="/users/signup">Sign Up Free</a>
<p>You need to signup to register for free classes</p>
```

Let's reload the app:



Welcome To AIoTDesign

Lorem ipsum dolor sit amet, consectetuer *adipiscing elit*, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation **ullamcorper suscipit lobortis** nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum.

You need to signup to register for free classes

Latest Classes

Step 24. Run MongoDB Community Edition

We will run the Command Prompt as an administrator and we'll go to your MongoDB bin directory. In this case, it's in C: drive, so we will run the `mongod` command. Then, we will run `mongo`.

1. **Create database directory**
   Create the data directory where MongoDB stores data. MongoDB's default data directory path is the absolute path **\data\db** on the drive from which you start MongoDB.

   From the **Command**, create the data directories:

```
cd C:\
md "\data\db"
```

2. **Start your MongoDB database**
   To start MongoDB, run `mongod.exe`.

```
"C:\Program Files\MongoDB\Server\4.2\bin\mongod.exe" --dbpath="c:\data\db"
```

   The `--dbpath` option points to your database directory.
   If the MongoDB database server is running correctly, the **Command** displays:

```
[initandlisten] waiting for connections
```

3. **Connect to MongoDB**
   To connect a `mongo.exe` shell to the MongoDB instance, open another **Command window** with and run:

```
"C:\Program Files\MongoDB\Server\4.2\bin\mongo.exe"
```

Step 25. Adding some classes

Now, we will create a database by saying `use aiot`, and we're going to create a couple of collections. We'll say `db.createCollection`. We want one called users and then we want one for our students, instructors, and classes:

```
> use aiot
switched to db aiot
> db.createCollection('users');
{ "ok" : 1 }
```

```
> db.createCollection('students');
{ "ok" : 1 }
> db.createCollection('instructors');
{ "ok" : 1 }
> db.createCollection('classes');
{ "ok" : 1 }
>
```

Now we want to add a couple of classes. To do that, we'll add the `db.classes.insert`. We now need a title. Let's say 'Intro to HTML5', description:'');. I'm just going to grab the text we have in the HTML 101 column in the app and paste that in and then an instructor:

```
> db.classes.insert({title:'Intro to HTML 5', description: 'Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis', instructor: 'Brad Traversy'});

WriteResult({ "nInserted" : 1 })
```

Now we have one class, let's go ahead and add another one. I'm going to change the name to John Doe and then we'll also change the title. Alright, so this one let's say Advanced PHP:

```
>db.classes.insert({title:'Advanced PHP', description: 'Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis', instructor: 'John Doe'});

WriteResult({ "nInserted" : 1 })
```

Then we'll do one more. So this one let's call Intro to Photoshop:

```
>db.classes.insert({title:'Intro to Photoshop', description: 'Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis', instructor: 'John Doe'});
WriteResult({ "nInserted" : 1 })
```

Now we should have three classes. So we can close out of that and what we need to do now is we need to create a class model.

Step 26. Creating a class model

So let's create a new folder called models and then inside there, we'll create a file called class.js. Let's include mongoose, and then we need a schema for our class, which I'm going to paste in:

```
var mongoose = require('mongoose');

// Class Schema
var ClassSchema = mongoose.Schema({
            title: {
                type: String
            },
            description: {
                type: String
            },
            instructor:{
               type:String
            },
            lessons:[{
               lesson_number: {type: Number},
               lesson_title: {type: String},
               lesson_body:{type: String}
            }]
});
```

We have a title description and instructor, and then we're also going to have lessons and this is going to be an array of objects and the object will have lesson number `title` and `body`. Then what we need to do is make this available outside, so we are going to paste this in:

```
var Class = module.exports = mongoose.model('Class', ClassSchema);
```

Now we're going to have a couple functions in our model.

Step 27. Fetch all classes

We're going to have one that's going to fetch all classes. So this one will say `module.exports.getClasses`. We want to set that equal to a `function`, and that's going to take `callback` and then also `limit` and then in the function what we'll do is say `Class.find` and we'll pass in callback and say `.limit`, whatever the limit that's passed in:

```
// Fetch All Classes
module.exports.getClasses = function(callback, limit){
    Class.find(callback).limit(limit);
}
```

Step 28. Fetch single classes

Then we need one to fetch single classes, so use `module.exports.getClassById`. That's obviously going to take an `id` and then a `callback`. Then in the function what we'll do is say `Class.findById` and pass in the `id` and the callback. So that's it for that:

```
// Fetch Single Class
module.exports.getClassById - function(id, callback){
   Class.findById(id, callback);
}
```

Let's save it and now we need to go to our route.

Step 29. Working on the GET home page route

So we're going to go to `routes` directory and then `index.js` and we want to work on this GET home page route. What we want to do is use the `class` model and then get all the classes and then pass to our view. So before we can do that, we have to create our class object here. We're going to set this to require, and we want to say ../ to go up one and then `models/class`:

```
var Class = require('../models/class');
```

Then in the `router.get` function, we can say `class.getClasses` and we want our function. That will take the error and classes, and then what we want to do is grab the `res.render` statement and put that below the `Class.getClasses` statement:

```
/* GET home page. */
router.get('/', function(req, res, next) {
   Class.getClasses(function(err, classes){
       res.render('index', { title: 'Express' });
   });
});
```

Then we just want to pass along the classes. Now, remember we have a limit we can set, so we're going to 3, like this:

```
       res.render('index', { classes: classes });
},3);
```

Alright so now the index view should have access to our classes.

We want to go to the partial `classes.handlebars` and we're going to get rid of two out of three of these divs. We just want to wrap this, say `{{#each classes}}`. Then, we want the title. We want the description and then for the link it's going to go `/classes/_id}}/details`:

```
    {{#each classes}}
        <div class="col_4">
        <h4>{{title}}</h4>
        <p>{{description}}</p>
        <a class="button" href="/classes/{{_id}}/details">View Class</a>
      </div>
    {{/each}}
```

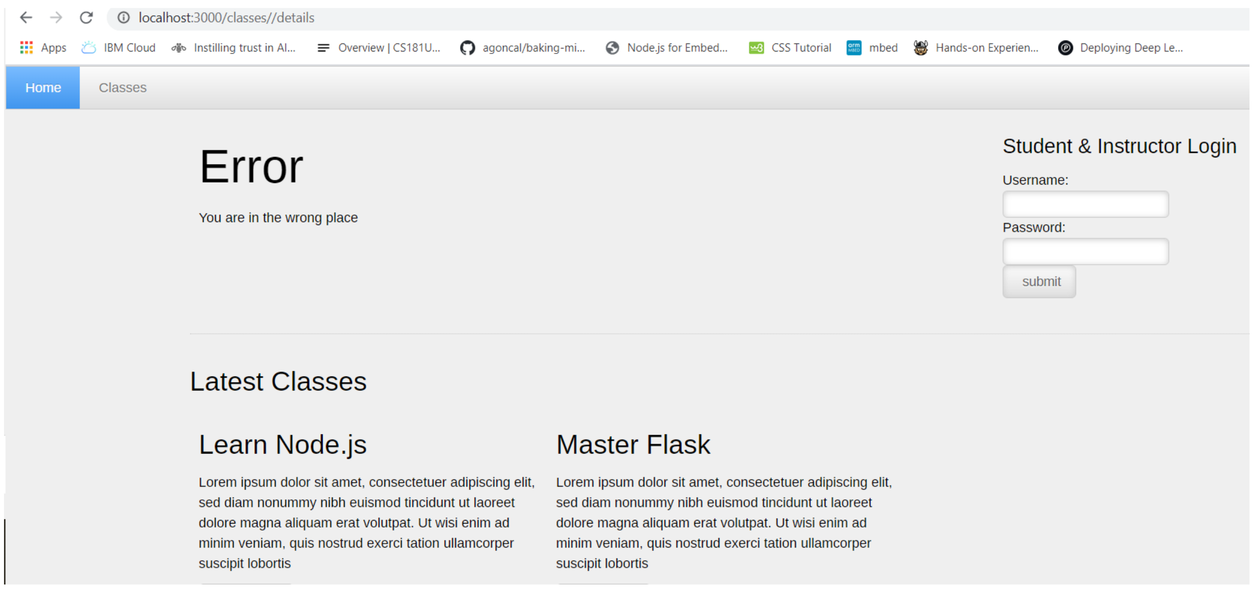
Let's go to `apps.js` and right under the mongoose variable, we're going to say `mongoose.connect`. Then in here, we'll say `mongodb://localhost/aiot` and then right under it, we're going to just say `var db = mongoose.connection`:

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/aiot');
var db = mongoose.connection;
```

So let's save that and let's restart. it's getting the classes. Now these classes are being fetched from the database. Now in the next part of this section, what I want to do is take care of the details page so we have a single class view. And then when we click on Classes, I want to list all the classes, just like we do on the index but without a limit. We'll do that in the next part.

Step 29. Fetching classes – part B

In this section, we're going to set up the **Classes** page


localhost:3000/classes//details
Apps
IBM Cloud
Instilling trust in AI...
Overview | CS181U...
agoncal/baking-mi...
Node.js for Embed...
CSS Tutorial
mbed
Hands-on Experien...
Deploying Deep Le...
Home
Classes
Error
You are in the wrong place
Student & Instructor Login
Username:
Password:
submit
Latest Classes
Learn Node.js
Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis
Master Flask
Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Here's so the details page. OK, so what we need to do is we need to create a new routes file

Step 30. Setting up new route file – classes.js

We're going to call this new routes file `classes.js`. Then what we'll do is copy everything from `index.js` and paste that in:

```
var express = require('express');
var router = express.Router();
var Class = require('../models/class');
/* Get home page */
router.get('/', function(req, res, next) {
    Class.getClasses(function(err, classes){
      res.render('index', { classes: classes });
    },3);
});

module.exports = router;
```

This is pretty much going to stay the same except we want to change the render to `classes/index`:

```
var express = require('express');
var router = express.Router();
var Class = require('../models/class');
// Classes Page
router.get('/', function(req, res, next) {
    Class.getClasses(function(err, classes){
       res.render('classes/index', { classes: classes });
    },3);
});
module.exports = router;
```

Let's save this file. Next, we're going to include the routes file in `app.js`. So what we need to do is add a classes variable and require it like this:

```
var users = require('./routes/users');
var classes = require('./routes/classes');
```

Delete the previous “`var usersRouter = require('./routes/users')`”. Then down below to `app.use`, we also need to add it below the ‘`app.use('/', indexRouter)`’ statement:

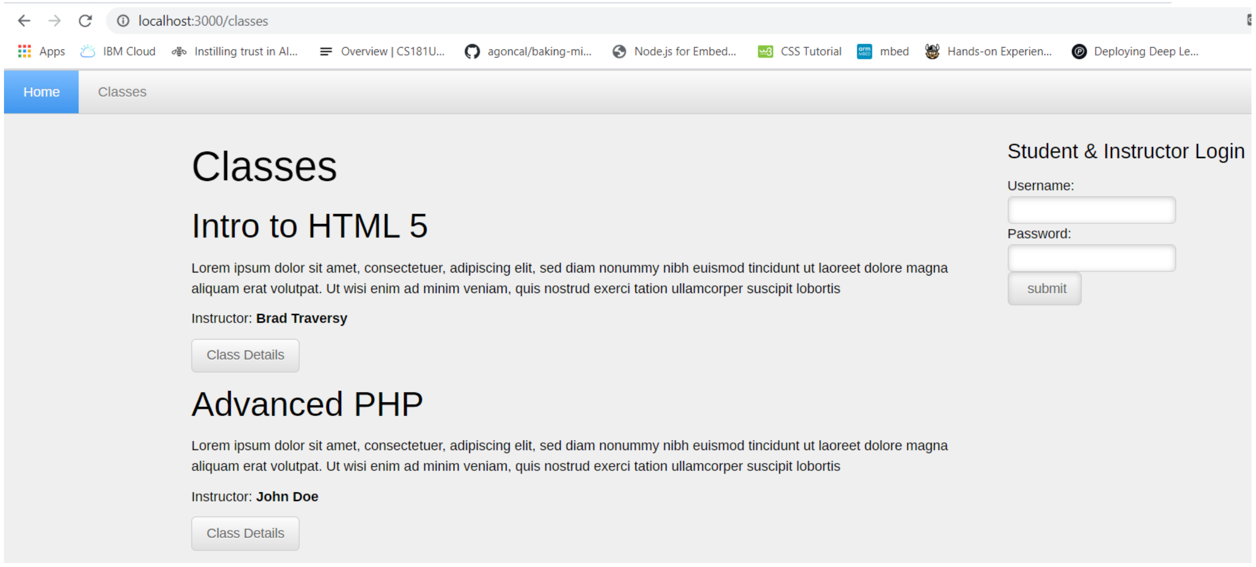
```
app.use('/', indexRouter);

app.use('/users', users);
app.use('/classes', classes);
```

Step 31. Creating the index.handlebars file for the classes page

Next, in our `views` folder we're going to create a new folder called `classes`. Then in here, we're going to create a new file called `index.handlebars`. We'll open that up and I'm going to paste this in:

```
<h2>Classes</h2>
{{#each classes}}
   <div class="block">
     <h3>{{title}}</h3>
     <p>
     {{description}}
     </p>
     <p>Instructor: <strong>{{instructor}}</strong></p>
     <a class="button" href="/classes/{{_id}}/details">Class Details</a>
   </div>
{{/each}}
```

Here, we're just looping through the classes. We're displaying a div with the class of block. We have a title description. We have the instructor and then a link to the details. Let's save that and then we'll go ahead and restart the server. Then click on **Classes**:



Classes

Intro to HTML 5

Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Instructor: **Brad Traversy**

Class Details

Advanced PHP

Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Instructor: **John Doe**

Class Details

Now we have a page with classes.

Step 32. Configuring classes.js for the class details page

For the details, what we'll do is go back to our `routes` and then `classes.js` and let's copy the **Class Page** code snippet and this will be **Class Details**:

```
// Class Details
router.get('/:id/details', function(req, res, next) {
   Class.getClassById([req.params.id],function(err, classes){
      res.render('classes/details', { class: classes });
   },3);
});
```

The route is going to go to `/:id/details`. Then in the `getClasses` object will be `getClassById` method. Then let's change this to `classname.` Then we also need to pass in a parameter just before the function object, which will be the `id`. We're going to use some brackets and then we're going to say `req.params.id`. Then in the render statement, we're just going to render `classes/details`. We'll then pass in `classname`. Now, we don't need the limit 3 so we can get rid of that. The resultant code should look like this:

```
// Class Details
router.get('/:id/details', function(req, res, next) {
   Class.getClassById([req.params.id],function(err, classname){
      res.render('classes/details', { class: classname });
   });
});
```

We can check for an error in both of the **Class Details** and **Class Page**. So let's just say `if(err) throw err` like this:

```
//Classes Page
router.get('/', function(req, res, next) {
    Class.getClasses(function(err, classes){
       if(err) throw err;
       res.render('classes/index', { classes: classes });
    },3);
});

// Class Details
router.get('/:id/details', function(req, res, next) {
   Class.getClassById([req.params.id],function(err, classname){
      if(err) throw err;
      res.render('classes/details', { class: classname });
   });
});
```

So we'll save that.

Step 32. Creating details.handlebars for the class details page

Now what we need to do is go to our `views/classes` folder and then create a new file, save it as `details.handlebars`. We'll put in an `h2` and this is going to be the `class.title`:

```
<h2>{{class.title}}</h2>
```

That's our h2. Now let's put in a paragraph and then this is going to be the instructor. Let's put a `<strong>` tag and say `Instructor` and then we can say `class.instructor`. Under that, let's put in the description:

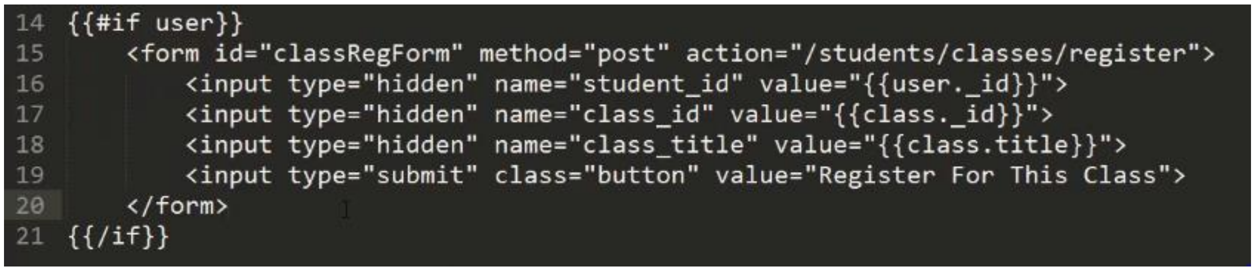
```
<h2>{{class.title}}</h2>
<p>
  <strong>Instructor: </strong> {{class.instructor}}
</p>
<p>
  {{class.description}}
</p>
```

Under the description, we're going to take the lessons and spit them out in a list, so we'll say `ul`. I'm going to give this a class of "`alt`" and then what we want to do is say `#each class.lessons`, which we don't have any now. The instructors will be able to add the lessons later. So let's end the each and in the each object, we want just list items. We should be able to get `lesson_title`:

```
<ul class="alt">
   {{#each class.lessons}}
      <li>{{lesson_title}}</li>
```

```
  {{/each}}
</ul>
```

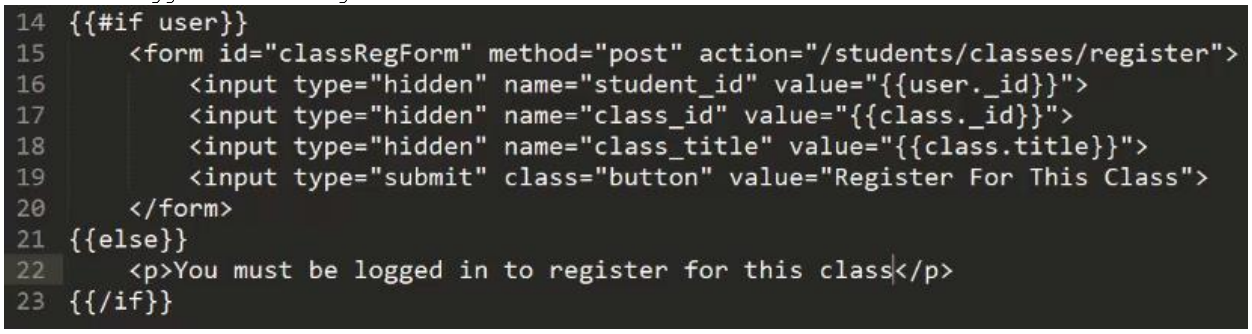
Under the lesson, we're going to want a form to register for the class. But in order to do that, we need the user to be logged in. Now, we don't have the login functionality yet but we will later. So, we'll add `if user`, then let's create a form, like this:



```
{{#if user}}
    <form id="classRegForm" method="post" action="/students/classes/register">
        <input type="hidden" name="student_id" value="{{user._id}}">
        <input type="hidden" name="class_id" value="{{class._id}}">
        <input type="hidden" name="class_title" value="{{class.title}}">
        <input type="submit" class="button" value="Register For This Class">
    </form>
{{/if}}
```

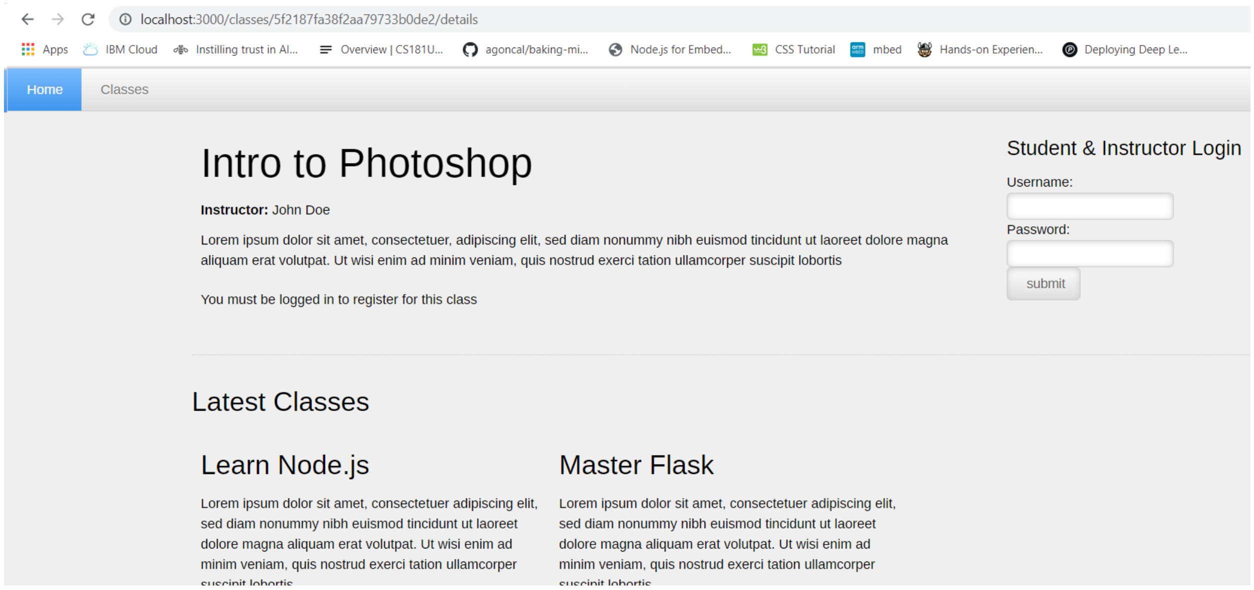
form `id="classRegForm"` will go to `students/classes/register`. We have some hidden inputs. This is going to be the `student id`, which we will be able to get with this once they're logged in. We have access to the `class ID`, the class title, and then a Submit button.

I'll now add an else statement. So just move this down and we'll say `{{else}}` and then we'll put a paragraph, "`You must be logged in to register for this class`":



```
{{#if user}}
    <form id="classRegForm" method="post" action="/students/classes/register">
        <input type="hidden" name="student_id" value="{{user._id}}">
        <input type="hidden" name="class_id" value="{{class._id}}">
        <input type="hidden" name="class_title" value="{{class.title}}">
        <input type="submit" class="button" value="Register For This Class">
    </form>
{{else}}
    <p>You must be logged in to register for this class</p>
{{/if}}
```

Let's save that and restart the server. If I click on **View Class**, it takes us to the details page:



localhost:3000/classes/5f2187fa38f2aa79733b0de2/details

Intro to Photoshop

Instructor: John Doe

Lorem ipsum dolor sit amet, consectetuer, adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

You must be logged in to register for this class

Student & Instructor Login

Username:

Password:

submit

Latest Classes

Learn Node.js

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

Master Flask

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis

We have the title, the instructor, and it's telling us we have to be logged in to register. That's exactly what we want. So in the next section, what we'll do is we'll start to work on the user registration and login.

Step 32. Registering users

In the last section, we made it so that we could fetch classes from the database and display them. So now what we want to do is work on the login and registration. So before we can do anything with login and registration we need to create a user model.

1. **Creating a user model**
   Under models, let's create a new file and save it as `user.js`. Now this is where we're going to keep the basic user information such as the `username` and `password` but we will also be creating a student model and then an instructor model that will hold things like their address and extended information. Let's just do this one really quick. We need Mongoose. We need bcrypt for password encryption and then we paste in the User Schema:

```
var mongoose = require('mongoose');
var bcrypt = require('bcryptjs');

// User Schema
var UserSchema = mongoose.Schema({
       username: {
              type: String
       },
       email: {
              type: String
       },
       password:{
              type:String,
              bcrypt: true
       },
       type:{
              type:String
       }
});
```

   This is very simple, we have username, email password, and then we also have a type, so the type is going to be whether they're an instructor or a student. The next line is we're going to assign the model to a variable and also make it available outside of this file:

```

var User = module.exports = mongoose.model('User', UserSchema);
```

   So the next thing we want some functions for the user model.

2. **Get User by Id**
   We want to get a single user. So, we're going to paste that in `getUserById`. It's going to take in an id and a callback and then we're just going to call `User.findById`:

```
// Get User By Id
module.exports.getUserById = function(id, callback){
       User.findById(id, callback);
}
```

3. **Get User by Username**
   The next function we want to add is `Get User by Username`. This is going to be the same thing, except we need to just have a query saying we want a certain username and then we're just using `findOne`:

```
// Get User by Username
module.exports.getUserByUsername = function(username, callback){
       var query = {username: username};
       User.findOne(query, callback);
}
```

   We'll have two different ones. One is going to be to create the instructor and one is going to be to create the student account.

4. **Create Student User**

   So the first one will say `Create Student User`. The code is shown as follows:

```
// Create Student User
module.exports.saveStudent = function(newUser, newStudent, callback){
       bcrypt.hash(newUser.password, 10, function(err, hash){
              if(err) throw errl
              // Set hash
              newUser.password = hash;
              console.log('Student is being saved');
              async.parallel([newUser.save, newStudent.save], callback);
       });
}
```

We're going to take the `bcrypt.hash` function and pass in the new user's password, which comes as the first argument in the function. You'll notice that we're also passing in `newStudent`. Then down here, we're setting the hash and then we're going to call this `async.parallel` because we want to save two different things. We want to save in the users table with `newUser.save` or the users collection and then also in the student collection with `newStudent.save`, and `async.parallel` allows us to do that.

5. **Create Instructor User**
   So now we want to do the same thing for instructors. Let me just paste that in the code for instructor:

```
// Create Instructor User
module.exports.saveInstructor = function(newUser, newInstructor, callback){
       bcrypt.hash(newUser.password, 10, function(err, hash){
              if(err) throw errl
              // Set hash
              newUser.password = hash;
              console.log('Instructor is being saved');
              async.parallel([newUser.save, newInstructor.save], callback);
       });
}
```

   This is literally the same exact thing except we're passing in a `newInstructor` here and we're saving it to the instructors collection.

6. **Compare Password**
   Then the last function we need is the `Compare password`. I am going to put that just below the `GET` user by username function:

```
// Compare password
module.exports.comparePassword = function(candidatePassword, hash,callback){
    bcrypt.compare(candidatePassword, hash, function(err, isMatch){
      if(err) throw err;
      callback(null, isMatch);
    });
}
```

   That's just taking in a candidate password and it's going to hash it and it's going to check to see if it's a match. We'll save that and that should do it for the user model.

Step 33. Configuring User Register

Now we want to go to `routes` and then `users.js` and we want to get rid of the `/*GET users listings*/` statement. This will be User Register. Then we want the / to go to `/register`. We'll get rid of the `res.send` statement, and we just want this to render the template. So we're going to say `res.render` and we want to render `users/register`:

```
// User Register
router.get('/register', function(req, res, next) {
  res.render('users/register');
});

module.exports = router;
```

We'll save that.

Step 34. Configuring the register.handlebars file

Now, down in `views` folder all we need to do is create a new folder called `users`. Inside there, we'll create a file called `register.handlebars`, and I'm going to paste the code in:

```
<h2>Create An Account</h2>
```

```
<h2>Create An Account</h2>
<form id="regForm" method="post" action="/users/register">
    <div>
        <label>Account Type</label>
        <select name="type">
           <option value="student">Student</option>
           <option value="instructor">Instructor</option>
        </select>
    </div>
    <br />
    <div>
       <label>First Name: </label><input type="text" name="firstname">
    </div>
       <br />
       <div>
       <label>Second Name: </label><input type="text"  name="secondame">
       </div>
       <br />
       <div>
       <label>Street Address: </label><input type="text" name="address">
       </div>
       <br />
       <div>
       <label>City: </label><input type="text" name="city">
       </div>
       <br />
       <div>
       <label>State: </label><input type="text" name="state">
       </div>
       <br />
       <div>
       <label>Zip: </label><input type="text" name="zip">
       </div>
       <br />
       <div>
       <label>Email Address: </label><input type="text" name="email">
       </div>
       <br />
       <div>
       <label>Username: </label><input type="text" name="username">
       </div>
       <br />
       <div>
    <label>Password: </label><input type="password" name="password">
       </div>
       <br />
       <div>
    <label>Password Confirm: </label><input type="password" name="repassword">
    </div>
       <br />
       <div>
 <button name="submit" type="button" id="btnvalidate">Signup</button>
       </div>
</form>
```

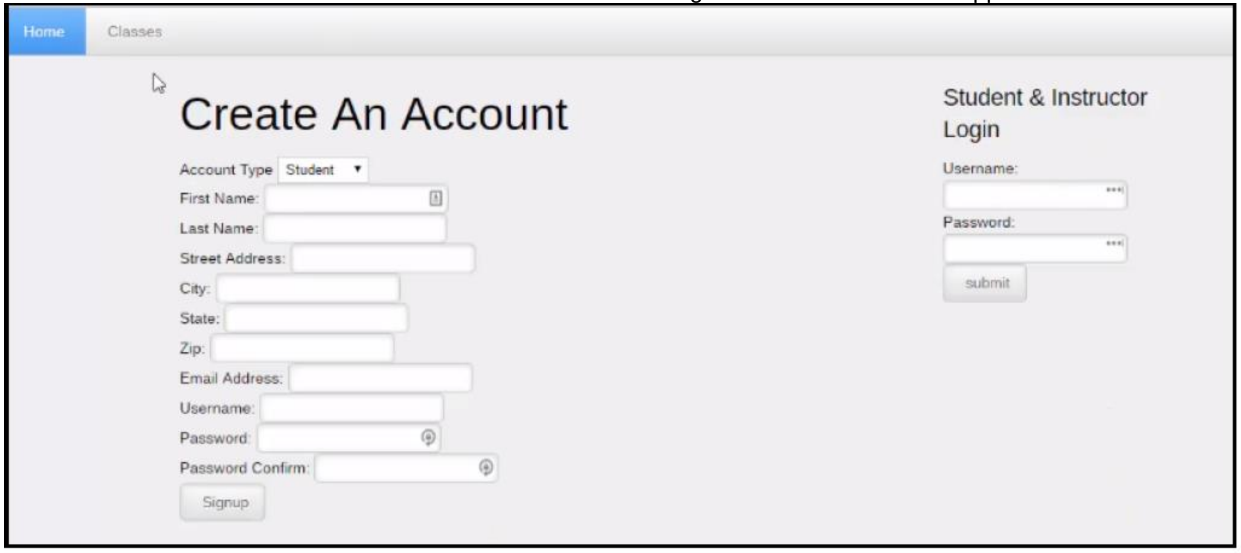
Now, this code is kind of a lot. It's a lot of different fields. So it's a form. We have the `id regForm`. It's going to go to `users/register`:

```
<h2>Create An Account</h2>
<form id="regForm" method="post" action="/users/register">
```

Now, the first option is to create an account type. You want to give this the name of type and then we have a `Student` or `Instructor`:

```
    <div>
        <label>Account Type</label>
        <select name="type">
           <option value="student">Student</option>
           <option value="instructor">Instructor</option>
        </select>
    </div>
```

OK, then we have first name, last name, street address, city, state, zip, email, username, and password. So let's save that and now we should be able to at least see the form. So let's go ahead and restart our app:



Create An Account

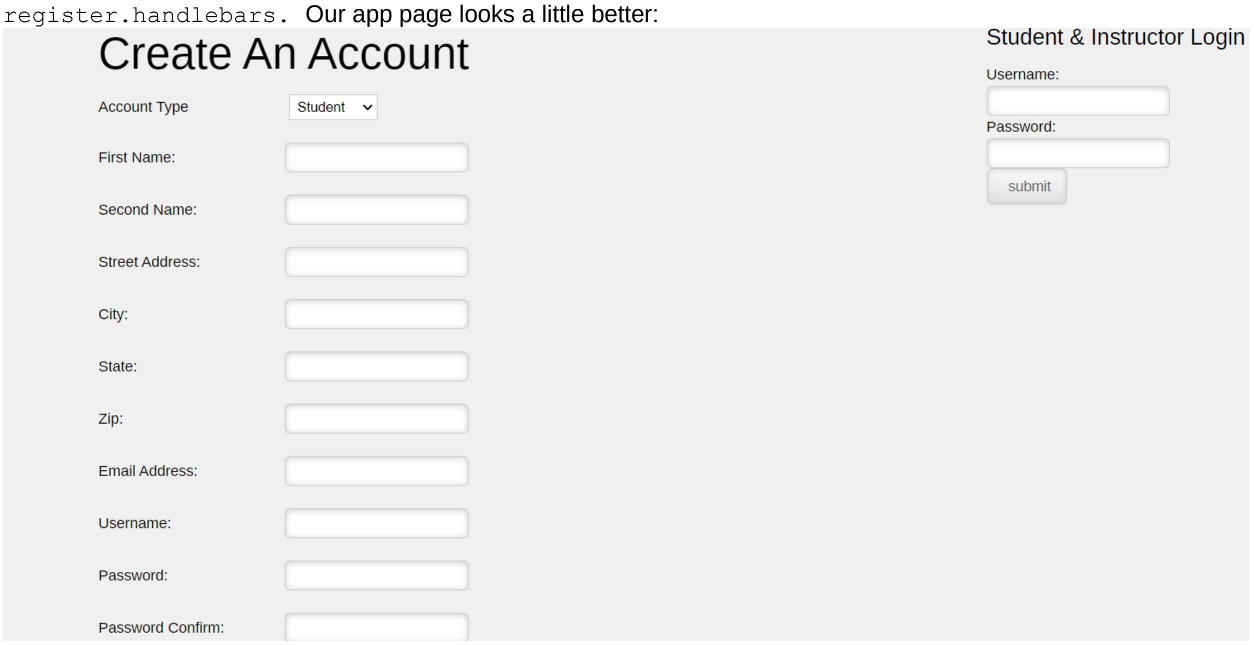
Account Type Student
First Name:
Last Name:
Street Address:
City:
State:
Zip:
Email Address:
Username:
Password:
Password Confirm:
Signup

Student & Instructor Login
Username:
Password:
submit

It doesn't look too good. we push all the inputs over a little bit, so what we can do is go to our public stylesheets and then `style.css` and we'll go all the way to the bottom. Let's just say form label. We want to display as an inline-block and let's set the width to 180:

```
form label{
    display:inline-block;
    width:180px;
}
```

We can close the `style.css` and I just want to put a line break in between each one of the `divs` in the `register.handlebars`. Our app page looks a little better:



Create An Account

Account Type Student
First Name:
Second Name:
Street Address:
City:
State:
Zip:
Email Address:
Username:
Password:
Password Confirm:

Student & Instructor Login
Username:
Password:
submit

To create the account, first we need to create our student and instructor models. Under `models` folder, we'll save this as `student.js` and then we'll save one as `instructor.js`.

Step 35. Creating the student model

Let's open up `student` and this is all stuff that we've done, so I'm going to just paste the code in. We'll create a schema here:

```
var mongoose = require('mongoose');

// Student Schema
var StudentSchema = mongoose.Schema({
	first_name: {
		type: String
	},
	last_name: {
		type: String
	},
	address: [{
		street_address:{type: String},
		city:{type: String},
		state:{type: String},
		zip:{type: String}
	}],
	username: {
		type: String
	},
	email: {
		type: String
	},
	classes:[{
		class_id:{type: [mongoose.Schema.Types.ObjectId]},
		class_title: {type:String}
	}]

});
```
So students are going to have a first name, last name, address, which is going to be an array, and it's going to have their own certain fields, username, email and then classes that they're registered to. Now what we're doing here is we're saying that the class id is going to map to the `ObjectId`. Then we're going to have a title and then here we're just basically creating the model and setting it to be available outside. Let's save that.

Step 36. Creating the instructor model
Then we're going to go to the `instructor` model, and we're going to paste that in and the same thing we're doing all the the user information and also the classes that this instructor has created:
```
var mongoose = require('mongoose');

// Instrucor Schema
var InstructorSchema = mongoose.Schema({
	first_name: {
		type: String
	},
	last_name: {
		type: String
	},
	address: [{
		street_address:{type: String},
		city:{type: String},
		state:{type: String},
		zip:{type: String}
	}],
	username: {
		type: String
	},
	email: {
		type: String
	},
	classes:[{
		class_id:{type: [mongoose.Schema.Types.ObjectId]},
		class_title: {type:String}
	}]
});
```
Then we'll make it available here. So we'll save that.
Now we're going to close these and we need to go back to our users route, in `routes` folder we want `users.js`.

Step 37. Configuring the users.js file

We're going to have to include `body-parser, passport`, along with `LocalStrategy`:

```
var express = require('express');
var router = express.Router();
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;
var expressValidator = require('express-validator');
router.use(expressValidator());
```

To implement the validation in the signup, after `var router = express.Router()` statement, we add:

```
var expressValidator = require('express-validator');
router.use(expressValidator());
```

Then we need to include all the different models because we're working with all of them. So I'm going to paste this in and we'll `require User, Student`, and `Instructor` models:

```
// Include User Model
var User = require('../models/user');
// Include Student Model
var Student = require('../models/student');
// Include Instructor Model
var Instructor= require('../models/instructor');
```

Then we need to catch the form when it's submitted. We want to create a post request to register, like this:

```
// Register User
router.post('/register', function(req, res, next) {

});
```

Now we'll grab all the values and put them into variables. So, I'm going to paste this in. There are quite a few variable as shown here:

```
// Register User
router.post('/register', function(req, res, next) {
  // Get Form Values
  var first_name = req.body.first_name;
  var last_name = req.body.last_name;
  var street_address = req.body.street_address;
  var city = req.body.city;
  var state = req.body.state;
  var zip = req.body.zip;
  var email = req.body.email;
  var username = req.body.username;
  var password = req.body.password;
  var password2 = req.body.password2;
  var type = req.body.type;
});
```

We're getting the first name, last name, all the address info, email, username, password, and password2 because we're going to validate it and then the type. The next thing we want to do is our form validation. I'm going to paste this stuff in:

```
// Form Validation
req.checkBody('first_name', 'First name field is required').notEmpty();
req.checkBody('last_name', 'Last name field is required').notEmpty();
req.checkBody('email', 'Email field is required').notEmpty();
req.checkBody('email', 'Email must be a valid email address').isEmail();
req.checkBody('username', 'Username field is required').notEmpty();
req.checkBody('password', 'Password field is required').notEmpty();
req.checkBody('password2', 'Passwords do not
match').equals(req.body.password);
```

These are all required, and then we do our password match here. After that, we're going to say `errors = req.validationErrors` and then `if(errors)`. If there are errors, then we're going to `res.render`. We want the same form to be rendered, users/register, and then we just want to pass along the errors and check for these errors:

```
const errors = req.validationErrors();
   if(errors){
     res.render('users/register', {
       errors: errors
     });
   } else {
   }
});
```

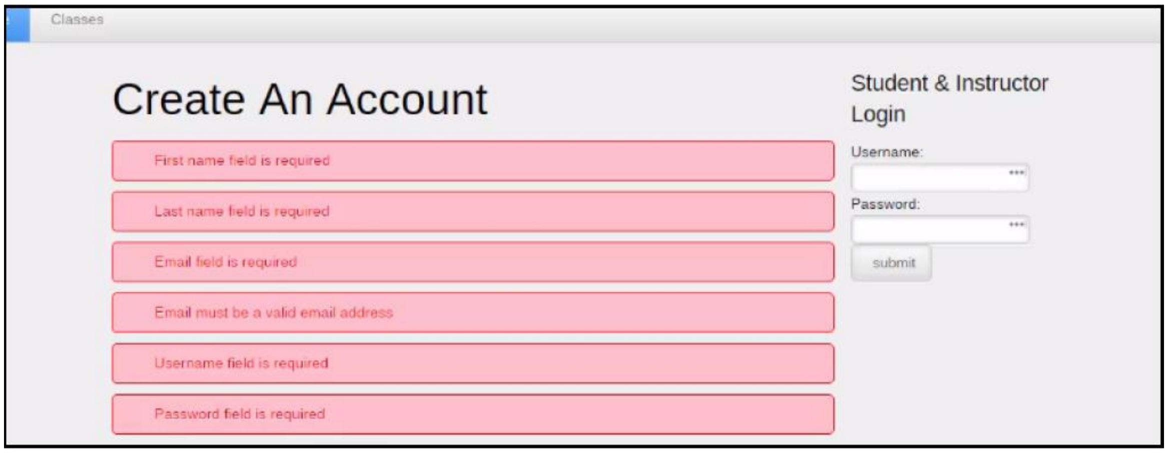
So let's go to `views/users`, and then our register page, and we're going to go ahead and just paste in the code, checking for errors:

```
<h2>Create An Account</h2>
{{#if errors}}
    {{#each errors}}
       <div class="notice error"><i class="icon-remove-sign iconlarge"></i> {{msg}}
       <a href="#close" class="icon-remove"></a></div>
  {{/each}}
{{/if}}

<form id="regForm" method="post" action="/users/register">
```

If there are errors, then we're going to loop through them and we're going to spit out a `div` and notice we have the class of error. So that's going to be kind of like a Bootstrap alert, and then we want to echo out the message. Let's save that and we'll test this out.

Step 37. Testing the app for the errors

Now, we'll restart the app. Then go to the free signup. If I click on **Sign Up**, we get a bunch of different errors, like this:



Create An Account

First name field is required

Last name field is required

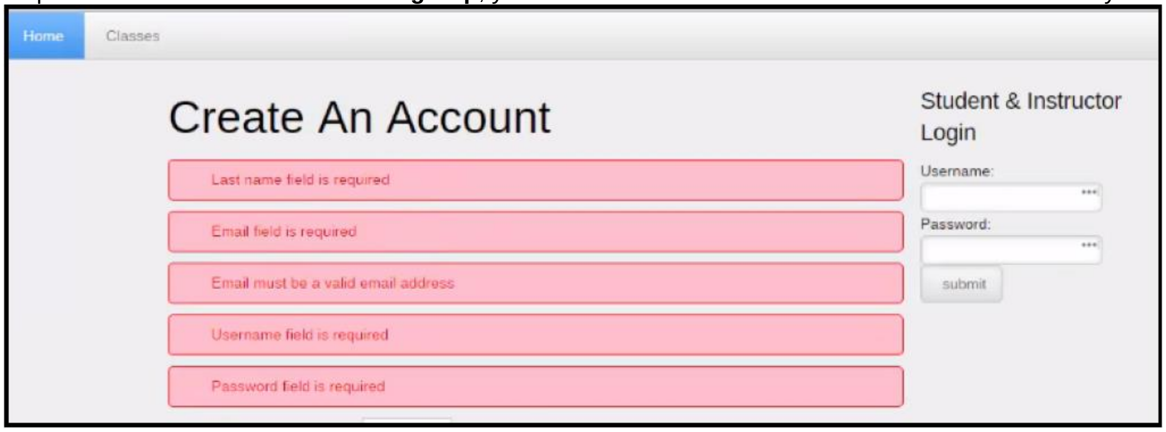
Email field is required

Email must be a valid email address

Username field is required

Password field is required

If I put in the first name and click on **Sign Up**, you can see that the error for the first name is not here anymore:



Create An Account

Last name field is required

Email field is required

Email must be a valid email address

Username field is required

Password field is required

So the form validation is working. Now we want each option to do what it is supposed to do when the form actually passes.

Step 38. Creating different objects in user.js for user collection

We need to create a couple of different objects in the else statement in `users.js`. So first we have the `newUser` object, which is going to go into the user collection:

```
  if(errors){
    res.render('users/register', {
      errors: errors
    });
  } else {
    var newUser = new User({
      email: email,
      username: username,
      password: password,
      type: type
    });
  }
});
```

Then, we want to test to see if the type is a student or an instructor. For this, I'm going to add an `if` statement, `if(type)`, remember we put the type into a variable, so, we'll add `if(type) == 'student'`. Then, we'll do something else, then we'll do something else; for now let's just `console.log(''Registering Instructor...')`; and then the if statement one is `'Registering Student...'`:

```
    if(type == 'student'){
       console.log('Registering Student...');
    } else {
       console.log('Registering Instructor...');
    }}
});
```

Step 38. Creating the new student object

For the student, we're going to have to create a new `student` object, so I'm going to paste the code in, just like we did with the new user, except we have all the fields that will go in the student collection:

```
   if(type == 'student'){
      console.log('Registering Student...');
      var newStudent = new Student({
      first_name: first_name,
      last_name: last_name,
      address: [{
        street_address: street_address,
        city: city,
        state: state,
        zip: zip
      }],
      email: email,
      username:username
});
```

After that, we're going to call `User.saveStudent`:

```
User.saveStudent(newUser, newStudent, function(err, user){
    console.log('Student created');
});
```

We're going to pass in the user and student object, and then just go ahead and create the student.

Step 38. Creating the new instructor object

I'm going to just copy what we have in the `if` statement, paste that in the `else` statement, and change the `newStudent` to `newInstructor`. Then we change the `Student` object to `Instructor`. We'll also change the `saveStudent` to `saveInstructor`:

```
   } else {
       console.log('Registering Instructor...');
       var newInstructor = new Instructor({
       first_name: first_name,
       last_name: last_name,
       address: [{
          street_address: street_address,
          city: city,
          state: state,
          zip: zip
       }],
       email: email,
       username:username
      });
        User.saveInstructor(newUser, newInstructor, function(err, user){
         console.log('Instructor created');
  });
```

Then outside this `if` statement, we'll just do `req.flash ('success'), '');` and then we'll redirect with `res.redirect('/'):`

```
  }
  req.flash('success_msg', 'User Added'); // this does not work here
  res.redirect('/');
  }
});
```

Let's save that and see what happens. Now, what I want to do is go to the `layout.handlebars`, go right above the body, and put in messages, like this:

```
<div class="grid">
<div class="col_12">
  <div class="col_9">
     {{{messages}}}
     {{{body}}}
  </div>
```

Let's go back to signup and register as an instructor. Let's try running the app again and restart it. Alright, so that redirected us to the first page.

Step 38. Logging in users

In this section, we're going to take care of the user login. Before we get to that, I want to address something. When we registered, in the last section, everything went well. The user got inserted, but we didn't get our message here, and we need to fix that. So, we're going to change `app.js` content just a little bit.

We go down to the `app.use` area and we're actually not even going to use `express-messages`. So, we can just get rid of the `res.local.messages`. Then we're going to just create two global variables: one for success messages and one for error messages. So, we're going to say `res.locals.success_msg` is going to equal `req.flash`, and then we are going to pass `success_msg`. We'll do the same thing for the error message. We'll change success to error. So, that's the `app.js` part:

```
// Global Vars
app.use(function (req, res, next) {
   res.locals.success_msg = req.flash('success_msg');
   res.locals.error_msg = req.flash('error_msg');
   next();
});
```

And remember, you have to have `Connect-Flash` and you have to have the middle layer initialized right above the `app.use` function:

```
// Connect-Flash
app.use(flash());
```

Now we go back into our `users` route in the `users.js` file, and go to our register post. Down at the bottom of our register post, we put this `req.flash` line. Now here I think it's just success. So, what you want to do is change it to `success_msg`:

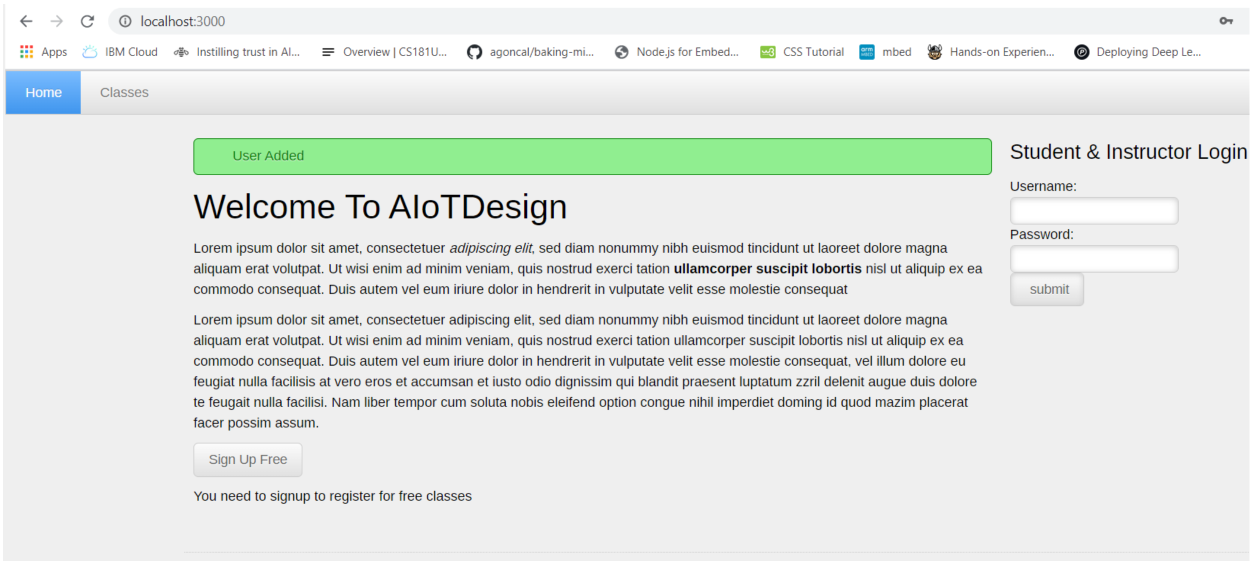
```
req.flash('success_msg', 'User Added');
```

That should actually work, so let's save that. Now we want to go to our main layout view, `layout.handlebars`. Right now, we just have messages and that's not what we want. What we want to do is check for the success message. So, I'm going to grab the code out of `registers.handlebars` where we have the `div` tag with the notice error. I'm going to grab the whole thing from the beginning `div` to the end `div`:

```
<div class="notice error"><i class="icon-remove-sign icon-large"></i>
{{msg}}
<a href="close" class="icon-remove"></a></div>
```

Before I paste that in, we're going to do an `if` statement. So, we're going to say `if success_msg`, and then down we'll end the `if`, and then in between we'll paste the above code:

```
{{#if success_msg}}
   <div class="notice error"><i class="icon-remove-sign icon-large"></i>{{msg}}
      <a href="#close" class="icon-remove"></a>
   </div>
{{/if}}
```
Now, what we want to do is change the above `msg` to `success_msg`. And since this is success, we want to change error to success. That'll make it green instead of red. Now if we want an error message, let's copy the above success message and then we're going to test for `error_msg`. We'll change this. We also want to change the class back to error:
```
{{#if error_msg}}
   <div class="notice error"><i class="icon-remove-sign icon-large"></i>{{msg}}
    <a href="#close" class="icon-remove"></a>
  </div>
{{/if}}
```
So, that'll test for both success and error messages. If there is any, it's going to output it as an alert. So, save that and just to test everything out, just to test the whole message system out, let's go ahead and register another user. And there we go, User Added as shown below.



User Added

Welcome To AIoTDesign

Lorem ipsum dolor sit amet, consectetuer *adipiscing elit,* sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation **ullamcorper suscipit lobortis** nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum.

Sign Up Free

You need to signup to register for free classes

Student & Instructor Login

Username:

Password:

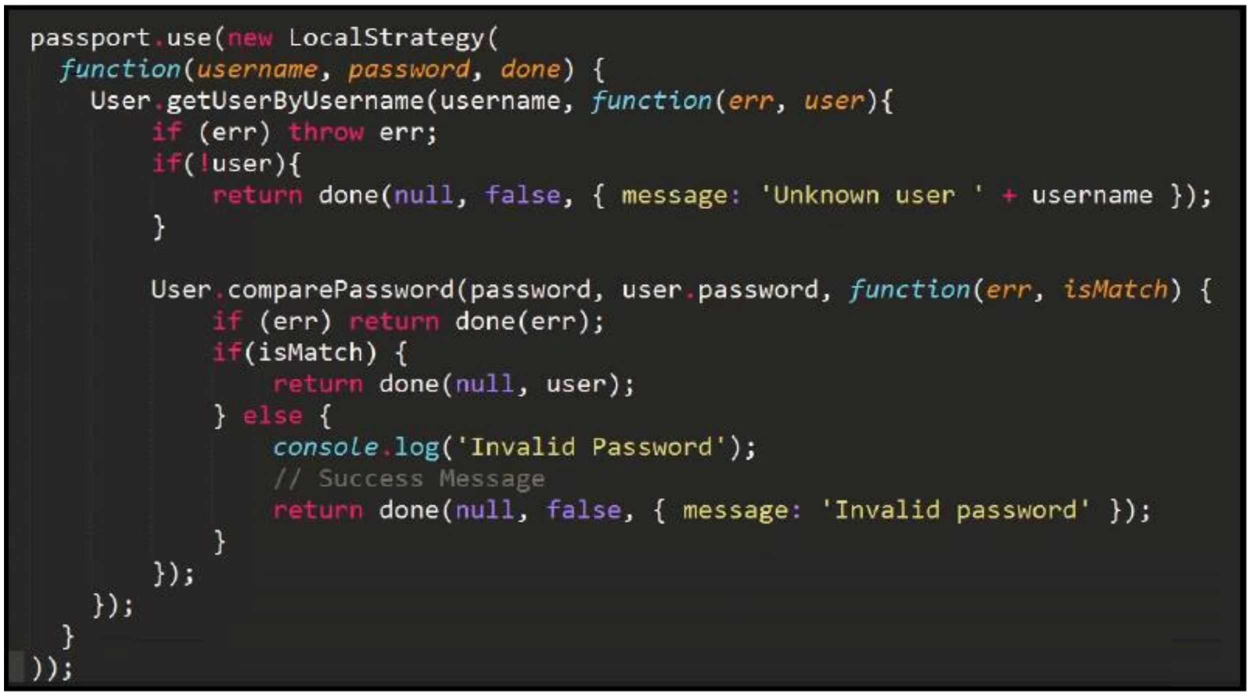
submit

Step 39. Setting up the Login page

If we look at the code for the login, which is in a partial, our form isn't going to have anything. So, we want to say `method= post` and the action will be `/users/login`. Let's go to our users route file (`routes/users.js`), we'll go right under the register, and I'm just going to copy the `router.get` method. We'll then say `router.post` and change `/register` to go to `/login`. Let's get rid of the `res.render`. Now, since we're using passport, we need to squeeze another parameter in the middle. We're going to say `passport.authenticate` and we want to pass in `failureRedirect.` We want it to go just to the **Home** page because that's where the login is.

Then we also want `failureFlash` and we're going to set that to true. Then when we're logged in, we'll set a flash message. So, let's say `req.flash` and pass in a `success_msg`, You are now logged in. Then we'll set the user type. We'll say `var usertype = req.user.type`. Then we'll say `res.redirect`. We want this to go to `/` and then we'll concatenate on `usertype`. We then say `usertype` and then we're going to concatenate an `s`.
We'll make it plural and then `/classes`. You'll see why in a little bit:
```
router.post('/login', passport.authenticate({failureRedirect:'/',
failureFlash:
true}), function(req, res, next){
req.flash('success_msg','You are now logged in');
var usertype = req.user.type;
res.redirect('/'+usertype+'s/classes');
});
```

Now what we need to do is we need to provide our local strategy. If you did this in the Node project, then this is pretty much the same thing. So, I'm going to paste it in:



```
passport.use(new LocalStrategy(
  function(username, password, done) {
    User.getUserByUsername(username, function(err, user){
        if (err) throw err;
        if(!user){
            return done(null, false, { message: 'Unknown user ' + username });
        }

        User.comparePassword(password, user.password, function(err, isMatch) {
            if (err) return done(err);
            if(isMatch) {
                return done(null, user);
            } else {
                console.log('Invalid Password');
                // Success Message
                return done(null, false, { message: 'Invalid password' });
            }
        });
    });
  }
));
```

So, we're saying `passport.use` and then we're saying new `LocalStrategy`. In the code, we have a function, which is going to call one of the user model functions, `getUserByUsername`. We're going to pass in the username and it's going to let us know. If there's no user, we're going to return done with false as the second parameter.
If there is a user, it'll keep going and we'll call `comparePassword`. We're going to pass in the candidate password and then the actual password and it's going to see if there's a match. If there is, we'll return done and pass in user. If there's not, then we're going to return done and pass in false. So, that's essentially what the code does.

Now, another thing we have to include here is the `serializeUser` and `deserializeUser` functions. So, `passport.serializeUser` is just returning done with the id. For `Deserialize`, we're going to call `getUserById`, which is in the user model. We already created these files `UserById` and `UserByUsername`:

```
passport.serializeUser(function(user, done) {
   done(null, user._id);
});

passport.deserializeUser(function(id, done) {
   User.getUserById(id, function (err,user) {
      done(err, user);
   });
});
```

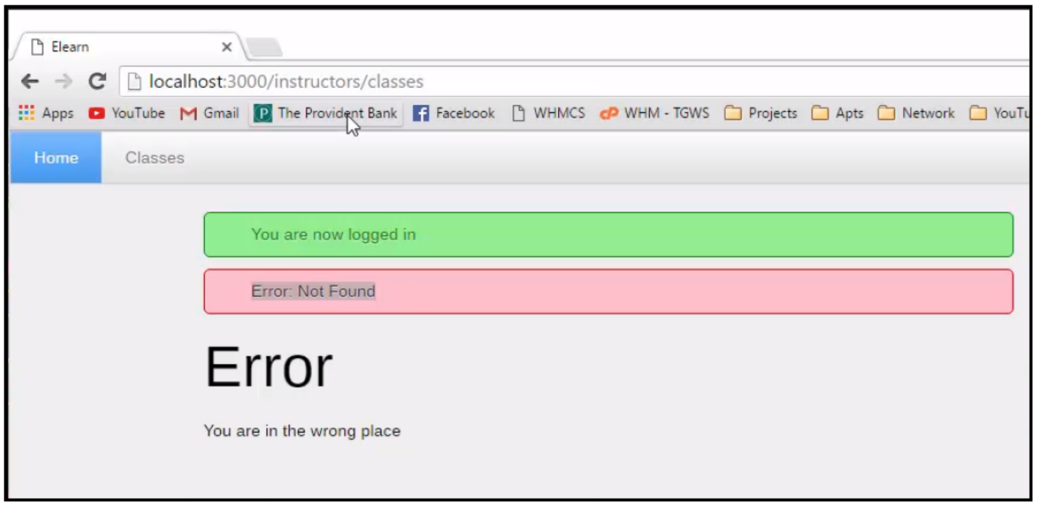
Make sure you have passport and local strategy present in the `users.js` file.

Now, the last thing we need to do with the login is handle the messaging. Because if we try to log in and we don't have the right password or username, we want it to tell us. So, what we need to do is go back to `app.js` and it's going to come at us in `req.flash` I'll copy the `res.locals.error` line and we're just going to take off the `_msg` so it's just error. That'll put whatever comes back to us in a global variable. So, we'll save that:

```
app.use(function (req, res, next) {
  res.locals.success_msg = req.flash('success_msg');
  res.locals.error_msg = req.flash('error_msg');
  res.locals.error = req.flash('error');
  next();
});
```

We'll go over to the layout and then we're going to test for that error global variable:

```
{{#if error_msg}}
   <div class="notice error"><i class="icon-remove-sign icon-large"></i>
   {{error_msg}}
   <a href="#close" class="icon-remove"></a>
</div>
{{/if}}

{{#if error}}
   <div class="notice error"><i class="icon-remove-sign icon-large"></i>
    {{error}}
    <a href="#close" class="icon-remove"></a>
   </div>
{{/if}}
```
We'll save that. Let's restart the server. And now if we go back and I try to just click on Submit, we get missing credentials. If I put in my username but the wrong password, we get invalid password. Then if we put in the correct info and click on Submit, we get You are now logged in:


Elearn
localhost:3000/instructors/classes
Apps YouTube Gmail The Provident Bank Facebook WHMCS WHM - TGWS Projects Apts Network YouTu
Home
Classes
You are now logged in

Error: Not Found

Error

You are in the wrong place

This `Error : Not Found` above has to do with the page not being in the route. You can look up in the `instructors/classes` URL and the reason we go to that error is because back in our `users.js` file and in the `res.redirect` route, we get into the user type. So, Brad is an instructor so he's going to `instructors/classes`. Now what we need to do is create this classes route within instructors and students.

Now, loading that page will take care of it a little bit, but for now, we want to finish our authentication functionality. We need a way to see if we're logged in or not, because now that we're logged in, we don't want the login form to show. We're also going to want other stuff to show or not to show. So, let's go to `app.js` and create something that's going to let us check to see if we're logged in or not. And we want to access this from anywhere, from everywhere. So, we're going to put it in `app.js`:
```
// Makes the user object global in all views
app.get('*', function(req, res, next) {
  // put user into res.locals for easy access from templates
  res.locals.user = req.user || null;
  if(req.user){
    res.locals.type = req.user.type;
  }
  next();
});
```
So, I'll paste it in and say `app.get` and then any page, or any route. So, this is going to work in any route. We're going to create a global variable called `user`. We're going to set it to `req.user` or null, if they're not logged in. If we're logged in, then we're going to get the type and we're going to put it in a global variable called `type`. So, we can check for this user value to see if they're logged in or not. Let's save that. And then we're going to go back to the `login.handlebars`, which is the partial:

```
{{#if user}}
  <h5>Welcome Back {{user.username}}</h5>
  <ul class="alt">
    <li><a href="/{{type}}s/classes">My Classes</a></li>
    <li><a href="/users/logout">Logout</a></li>
  </ul>
{{else}}
  <h5>Student & Instructor Login</h5>
  <form method="POST" action="/users/login">
    <label> Username: </label>
    <input type="text" name="username">
    <label> Password: </label>
    <input type="password" name="password">
      <div>
    <input type="submit" class="button" value="Login">
      </div>
  </form>
{{/if}}
```

The above code is checking for the `user` variable. So, if user, that means that they're logged in. If they are, then we want to just say Welcome Back and then their username. Then here we're going to have two links. One goes to My `Classes`. This has that type `s/classes` and then also a logout link. And since we restart the server, we have to log back in.

So, let's go to our `users.js` route and we're just going to paste that in. We're using a GET request to `/logout` and all we need to do is call this `req.logout` and that'll do everything for us. And then we want to set a message, except this should be success_msg and then that should be fine. So, let's save that and we're going to have to restart:

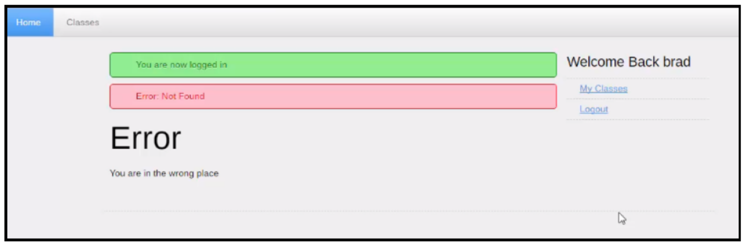
```
// Log User Out
router.get('/logout', function(req, res){
  req.logout();
  //Success Message
  req.flash('success_msg', "You have logged out");
  res.redirect('/');
});
```

Let's go back and log in. Now if I click on Logout, it brings us back. It tells us we've logged out.

In the next section, we're going to start to take care of the classes, the students registering classes, and so on and so forth.

Step 40. The Instructor and Student classes

In the last section, we made it so that we could log in and log out. So, what I want to do now is work on **Classes**; displaying them, displaying the currently logged in users classes, as well as being able to register for them. So, right now I can go to the app and log in by typing my credentials. This page created yet so we're going to get an error:



You are now logged in

Error: Not Found

Error

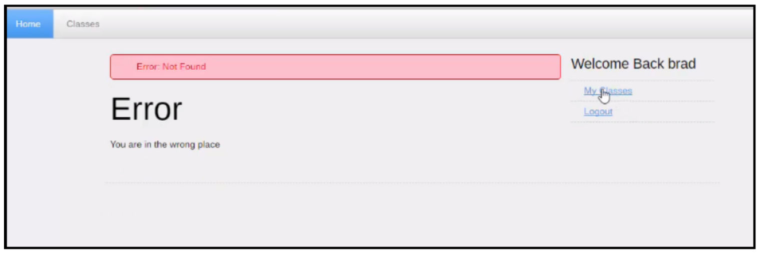
You are in the wrong place

Welcome Back brad

My Classes

Logout

The same is the case if I click on My Classes:



Error: Not Found

Error

You are in the wrong place

Welcome Back brad

My Classes

Logout

It's going to go to `instructors/classes` again. So, what we'll do is create two more routes: one for `students` and another for `instructors`. But before we do that, let's add it to our `app.js`.

We want to go where we have the rest of our routes and just add those in for students and instructors, right below the classes route:

```
var classes = require('./routes/classes');
var students = require('./routes/students');
var instructors = require('./routes/instructors');
```

Then we also need to include them in the `app.use` statements.

```
app.use('/students', students);
app.use('/instructors', instructors);
```

Now let's go to routes and create those two new files. One is `students.js` and the other is instructors.js.

Step 41. Configuring the student and instructor route

Let's open up `instructors` and `students` files. So, in `students`, I'm going to paste in code bit by bit. First of all, we'll include `express` and the `Router`. We'll also include all the other models:

```
var express = require('express');
var router = express.Router();

Class = require('../models/class');
Student = require('../models/student');
User = require('../models/user');
```

We're going to handle the `students /classes` route. So, let's add `router.get` and say `/classes` as the first argument and then a function as the second argument. In the function, we'll add the `req, res`, and next arguments like this:

```
router.get('/classes', function(req, res, next){

});
```

Then we're going to call the `getStudentByUsername`. Before that, we have to create it in the student model. To create this, let's go all the way down to the bottom and I'm going to just paste the code in. It's identical to the `getUserByUsername`, except it's for students:

```
module.exports.getStudentByUsername = function(username, callback){
   var.query = {username: username};
   Student.findOne(query, callback);
}
```

Similarly, we'll create the `getInstructorByUsername` function for the `instructors` route, like this:

```
module.exports.getInstructorByUsername = function(username, callback){
   var.query = {username: username};
   Instructor.findOne(query, callback);
}
```

So, let's save the `student` and the `instructor` models. Then, we'll go back to our `students` route. Now, in the `students` route, in the `router.get` statement, we're going to take that `Student` object and call `getStudentByUsername`. Then in here we're going to pass in `req.user.username`:

```
router.get('/classes', function(req, res, next){
    Student.getStudentByUsername(req.user.username);
});
```

So, this is how we can get the `username` of the currently logged in user. And then second parameter will be a function, which is going to take error and student as arguments. Then in the function, let's check for the error. Then we're going to say `res.render students/classes` and we'll pass in the student. At the bottom of this we have to have `module.exports = router`:

```
router.get('/classes', function(req, res, next){
   Student.getStudentByUsername(req.user.username, function(err, student){
     if(err) throw err;
     res.render('students/classes', {student: student});
   });
});
module.exports = router;
```

Now, I'm going to do the same thing in the `instructors` route. I'm going to copy the code from the `students` route and paste it into the `instructors` file. And we'll replace the student object with the instructor object:

```
var express = require('express');
var router = express.Router();

Class = require('../models/class');
Student = require('../models/student');
User = require('../models/user');

router.get('/classes', function(req, res, next){
  Student.getStudentByUsername(req.user.username, function(err, student){
    if(err) throw err;
    res.render('students/classes', {student: student});
  });
});
module.exports = router;
```

Let's save the file.

Step 42. Configuring the classes.handlebars files in student and instructor folders

So, now we're going to go into our `views` folder and we need to create two new folders, one is going to be students and the other one is going to be instructors. Inside those, we're going to have a file called `classes.handlebars`. So, we'll start with students `classes.handlebars`. We're going to paste this in:

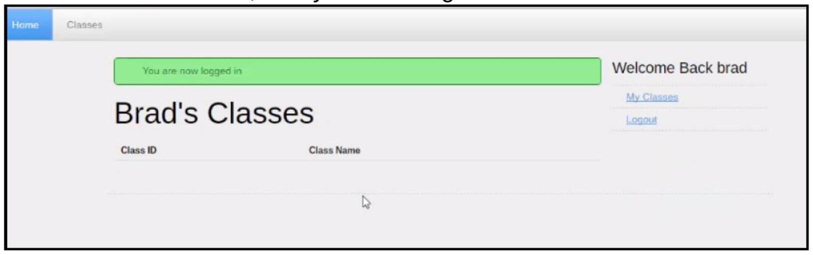
```
<h2>{{student.first_name}}'s Classes</h2>
<table cellspacing="0" cellpadding="0">
<thead>
<tr>
<th>Class ID</th>
<th>Class Name</th>
<th></th>
</tr>
</thead>
<tbody>
{{#each student.classes}}
<tr>
<td>{{class_id}}</td>
<td>{{class_title}}</td>
<td><a href="/classes/{{class_id}}/lessons">View Lessons</a></td>
</tr>
{{/each}}
</tbody>
</table>
```

Basically, we just have the user's first name, then classes, and then a table with a list of each class and a link to view the lessons. So, let's save that and then we'll go to our instructors one:

```
<h2>{{instructor.first_name}}'s Classes</h2>
<table cellspacing="0" cellpadding="0">
<thead>
<tr>
<th>Class ID</th>
<th>Class Name</th>
<th></th>
</tr>
</thead>
<tbody>
{{#each instructor.classes}}
<tr>
<td>{{class_id}}</td>
<td>{{class_title}}</td>
<td><a href="/classes/{{class_id}}/lessons">View
Lessons</a></td>
</tr>
{{/each}}
</tbody>
```

```
</table>
```

And this is pretty much the same exact thing except for instructors. So, we'll save that. So, let's see what happens if we reset the server. So, now you see we get Brad's Classes:


Home Classes

You are now logged in

Brad's Classes

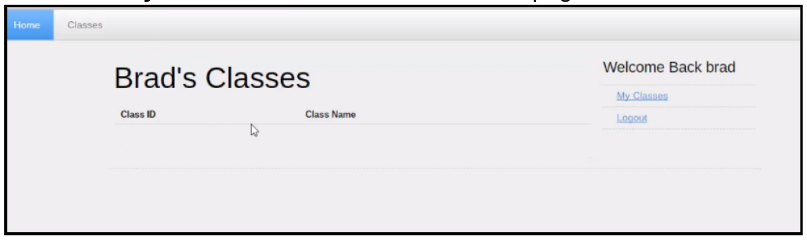
Class ID	Class Name

Welcome Back brad

My Classes

Logout

If I click on **My Classes**, we see almost the same page:


Home Classes

Brad's Classes

Class ID	Class Name

Welcome Back brad

My Classes

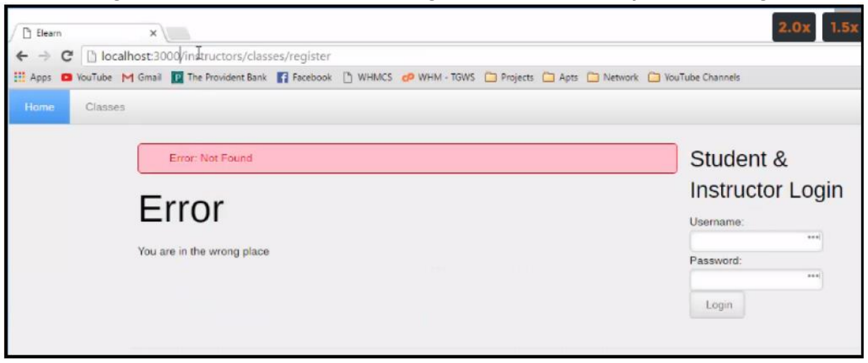
Logout

Step 43. Making the link to register dynamic

If we look in `app.js`, in the `app.get` request, we're setting the type as a global variable. So, we can use that in the actual in the link to register. So, let's go to `views/classes` folder and then the `details.handlebars` file. in this file, in the `if user` form, we have `students`. We can get rid of that and put in `{{type}}` and then just an `s`. So, let's save that file:

```
{{#if user}}
    <form id="classRegForm" method="post" action="/{{type}}s/classes/register">
           <input type="hidden" name="Student_id" value="{{user._id}}">
              <input type="hidden" name="class_id" value="{{class._id}}">
              <input type="hidden" name="class_title" value="{{class.title}}">
              <input type="submit" class="button" value="Register For This Class">
       </form>
{{else}}
    <p>You must be logged in to register for this class</p>
{{/if}}
```

Now if we go back to the server and we go view class, now you'll see it goes to `nstructors/classes/register`:


Elearn
localhost:3000/instructors/classes/register
Apps YouTube Gmail The Provident Bank Facebook WHMCS WHM - TGWS Projects Apts Network YouTube Channels
2.0x 1.5x
Home Classes

Error: Not Found

Error

You are in the wrong place

Student & Instructor Login

Username:

Password:

Login

Step 44. Making the instructor to able to register

Now, let's go back to our `instructors` route. Here, we want the instructor to be able to register. So, I'm going to paste the code in:

```
router.post('/classes/register', function(req, res){
  info = [];
  info['instructor_username'] = req.user.username;
  info['class_id'] = req.body.class_id;
  info['class_title'] = req.body.class_title;
  Instructor.register(info, function(err, instructor){
    if(err) throw err;
    console.log(instructor);
  });
  req.flash('success_msg', 'You are now registered to teach this class');
  res.redirect('/instructors/classes');
});
```

So, what we're doing here is `router.post classes/register`. Then we're going to initialize an array called `info`. In that array we're going to put the username, which comes from the currently logged in user. We're going to put in the class ID, which comes from the form. Now, when I say form I just mean the button, the `Register For The Class` button. This button is actually a form that we created if we look into our details.

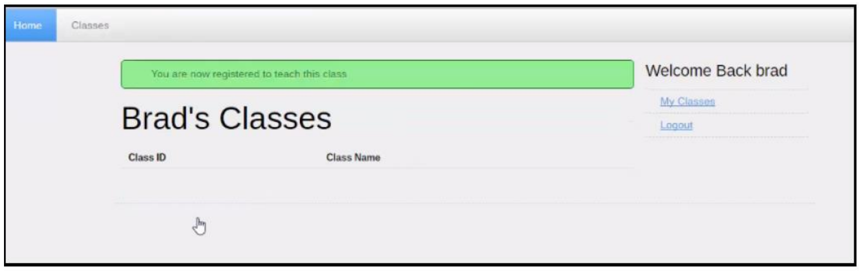
So, the class id is actually in the `details.handlebars` as a hidden input, along with the student ID and the class title. So, back to the instructor route. We get the class title in there as well. And then we'll call `instructor.register`. We're going to create this register in the instructor model. So, it'll register us and then it's going to redirect, setting out a flash message. And then we're going to redirect to our `Classes` page, `instructors/classes`. So, let's save that and let's go to the `instructor` model. Here at the bottom, we're going to
paste in this register function like this:

```
// Register Instructor for Class
module.exports.register = function(info, callback) {
  instructor_username = info['instructor_username'];
  class_id = info['class_id'];
  class_title = info['class_title'];
  var query = {username: instructor_username};
  Instructor.findOneAndUpdate(
    query,
    {$push: {"classes": {class_id: class_id, class_title: class_title}}},
    {safe: true, upsert: true},
    callback
  );
}
```

So, what we're doing is, remember, we passed in the `info` array. We're just setting those values to variables. Then we're creating a query variable. We're going to look for the username of the user that's logged in. Then we're going to call `instructor.findOneAndUpdate`, and we're going to pass in the query here. We're pushing the class that the instructor is registering with. We're pushing it on to the classes value inside the instructors collection. So, that's what that does. So, let's save the file and let's see what happens.

Step 45. Testing the instructor registration to the classes

Let's go to the server and login. Then let's go to **Classes** and register for the class:



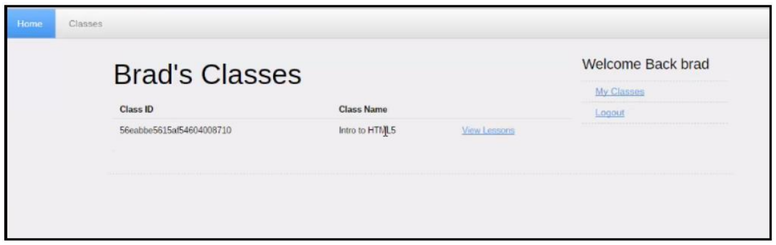
You are now registered to teach this class

Brad's Classes

Welcome Back brad

Class ID	Class Name

You are now registered for this class, to teach this class. It's also showing up in the My Classes link:

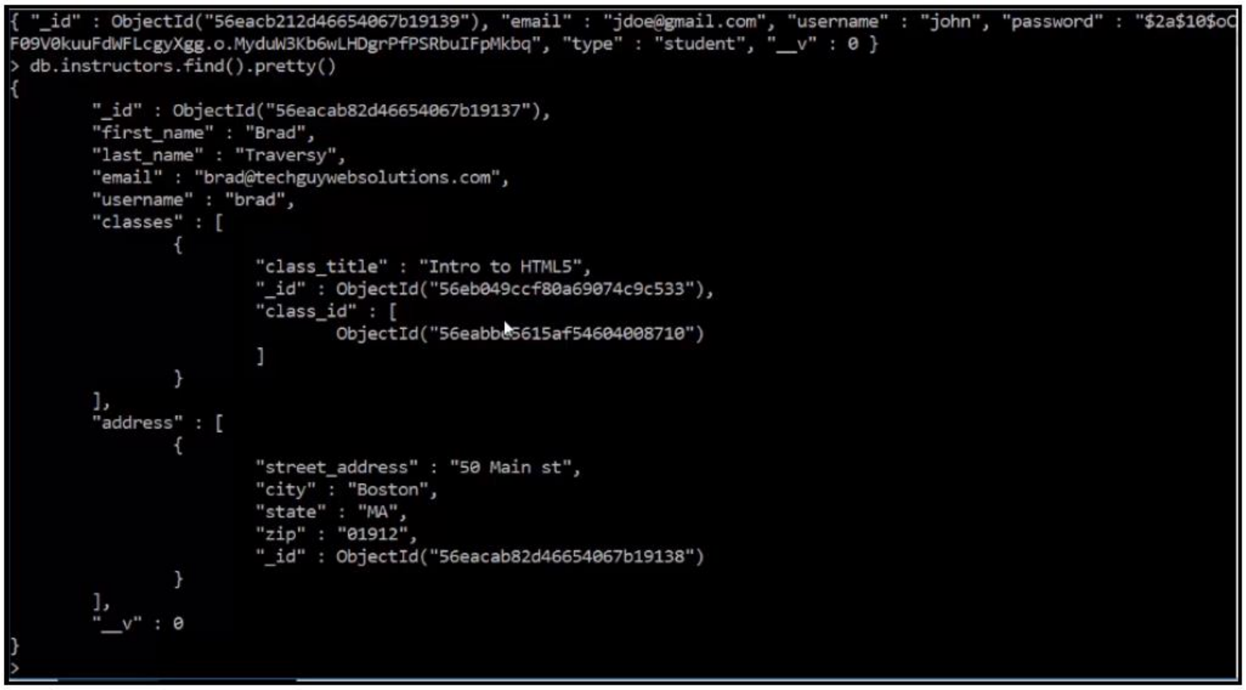


Brad's Classes

Welcome Back brad

Class ID	Class Name	
56eabbe5615af54604008710	Intro to HTML5	View Lessons

Now, if we look at the instructor record in the collection, we should have a classes value. So, let's just check that out. So, let's say `db.instructors.find.pretty`:

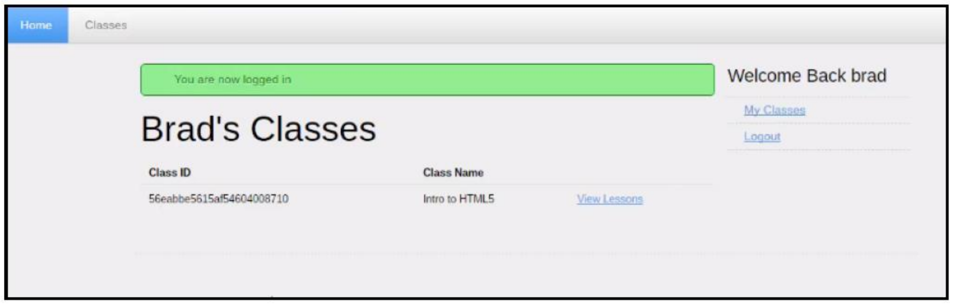


```
{ "_id" : ObjectId("56eacb212d46654067b19139"), "email" : "jdoe@gmail.com", "username" : "john", "password" : "$2a$10$oC
F09V0kuuFdWFLcgyXgg.o.MyduW3Kb6wLHDgrPfPSRbuIFpMkbq", "type" : "student", "__v" : 0 }
> db.instructors.find().pretty()
{
        "_id" : ObjectId("56eacab82d46654067b19137"),
        "first_name" : "Brad",
        "last_name" : "Traversy",
        "email" : "brad@techguywebsolutions.com",
        "username" : "brad",
        "classes" : [
                {
                        "class_title" : "Intro to HTML5",
                        "_id" : ObjectId("56eb049ccf80a69074c9c533"),
                        "class_id" : [
                                ObjectId("56eabbe5615af54604008710")
                        ]
                }
        ],
        "address" : [
                {
                        "street_address" : "50 Main st",
                        "city" : "Boston",
                        "state" : "MA",
                        "zip" : "01912",
                        "_id" : ObjectId("56eacab82d46654067b19138")
                }
        ],
        "__v" : 0
}
>
```

And then let's look at Brad. Now, you can see in classes I now have Intro to HTML5. This has an id, it also has the class id.
In the next section, we'll do this same kind of thing for students so that students can register for classes.

Step 46. Class lessons – the last section

We're almost there! There's one last piece of functionality that we need and that is for lessons. We need to be able to add them and show them. I'm logged in the classes as an instructor:



You are now logged in

Brad's Classes

Class ID	Class Name	
56eabbe5615af54604008710	Intro to HTML5	View Lessons

Now, I want to have a link beside View Lesson page that says Add Lesson

Step 47. Setting up the Add Lesson link in the Classes tab

We're going to go to the `views/instructors/classes.handlebars` file. In this file, I'm going to create another `<th>` in the `<thead>`, and then we want another td `<tbody>`, like this:

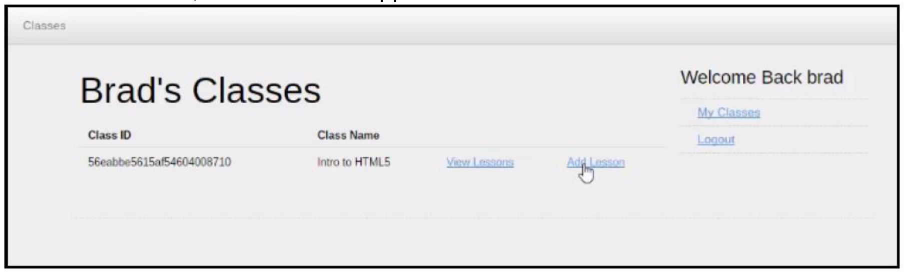
```
<thead>
  <tr>
    <th>Class ID</th>
    <th>Class Name</th>
    <th></th>
    <th></th>
  </tr>
</thead>
<tbody>
  {{#each instructor.classes}}
  <tr>
```

```
    <td>{{class_id}}</td>
    <td>{{class_title}}</td>
    <td><a href="/classes/{{class_id}}/lessons">View Lessons</a></td>
    <td><a href="/instructors/classes/{{class_id}}/lessons/new">Add Lesson</a></td>
  </tr>
  {{/each}}
</tbody>
```

Now, in `<td>`, I am going to add `Add Lesson` in place of `View Lesson`. We want this to go is `instructors/classes/` and then whatever the id and `/lessons/new`:

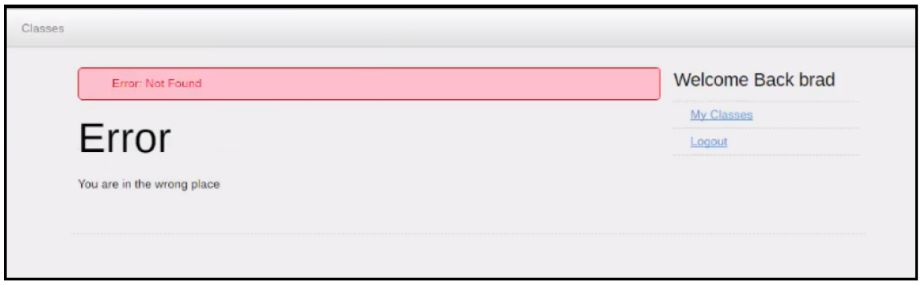
```
<td><a href="/instructors/classes/{{class_id}}/lessons/new">Add
Lesson</a></td>
```

Let's save this file, and reload the app:



Class ID	Class Name		
56eabbe5615af54604008710	Intro to HTML5	View Lessons	Add Lesson

So, now we have **Add Lesson**. If I click on the **Add Lesson** link, we're going to get an error:



Error: Not Found

Error

You are in the wrong place

We need to create this route.

Step 48. Creating route for the Add Lesson link

We'll go into `routes` and then `instructors.js`. And we'll go down to the bottom in this file. I'm actually just going to copy the `router.get` function and add it to the bottom:

```
router.get('/classes', function(req, res, next){
  Instructor.getInstructorByUsername(req.user.username, function(err,instructor){
  if(err) throw err;
    res.render('instructors/classes', {instructor: instructor});
  });
});
```

This will be `router.get`. Then, we go to `classes/:id/lessons/new`. Now, we're going to do is render the form, so I'm going to the render statement and then cut the remaining lines like this:

```
router.get('/classes/:id/lessons/new', function(req, res, next){
   res.render('instructors/classes', {instructor: instructor});
});
```

And we want to render `instructors/newlesson` and we can say `class_id` and set that to request, `req.params.id` like this:

```
router.get('/classes/:id/lessons/new', function(req, res, next){
    res.render('instructors/newlesson', {class_id:req.params.id});
});
```

Let's save the file.

Step 49. Configuring newlesson.handlebars

We'll go to the `views/instructors`, and then we want to create a new file. We'll save it as `newlesson.handlebars`. And let's copy the register form and paste it in this file:

```
<h2>Create An Account</h2>
{{#if errors}}
   {{#each errors}}
      <div class="notice error"><i class="icon-remove-sign iconlarge">
      </i> {{msg}}
      <a href="#close" class="icon-remove"></a></div>
   {{/each}}
{{/if}}
<form id="regForm" method="post" action="/users/register">
<div>
  <label>Account Type</label>
  <select name="type">
    <option value="student">Student</option>
    <option value="instructor">Instructor</option>
  </select>
</div>
<br />
<div>
```

It's going to be a lot smaller, so let's edit it. This will just say Add Lesson as the heading. We'll just get rid of error code snippet. Then the action is going to be `/instructors/classes/{{class_id}}/lessons/new`:

```
<h2>Add Lesson</h2>
<form id="regForm" method="post"
action="/instructors/classes/{{class_id}}/lessons/new">
```

Noe, let's get rid of `Account Type` div and we're going to have Lesson Number in the first name div. So, that's going to be a text field and then the name is going to be `lesson_number` and you can get rid of the value:

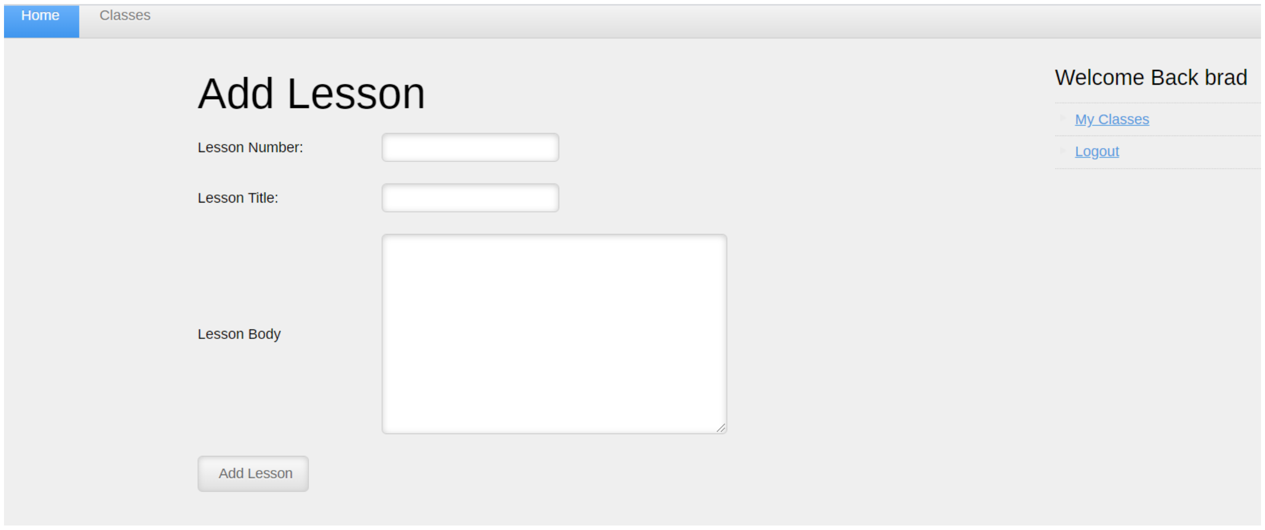
```
<h2>Add Lesson</h2>
<form id="regForm" method="post"
action="/instructors/classes/{{class_id}}/lessons/new">
<div>
  <label>Lesson Number: </label>
  <input type="text name="lesson_number">
<div><br />
```

Then what I'm going to do is just copy that and then get rid of the rest of the divs in the file except for the Submit button at the bottom. Then we'll just paste in the code two more times. The second div will be the Lesson Title, and the last one will be the Lesson Body. In the Lesson Body, we're going to be have text area so we can get rid of the input type:

```
<h2>Add Lesson</h2>
<form id="regForm" method="post"
action="/instructors/classes/{{class_id}}/lessons/new">
<div>
  <label>Lesson Number: </label>
  <input type="text name="lesson_number">
<div><br />
<div>
  <label>Lesson Number: </label>
  <input type="text name="lesson_number">
<div><br />
<div>
  <label>Lesson Number: </label>
  <input type="text name="lesson_number">
<div><br />
<div>
  <label>Lesson Number: </label>
  <input type="text name="lesson_number">
<div><br />
```

```
<div>
  <input type="submit" value="Add Lesson">
</div>
```

So, let's save that and make sure that the form actually shows up. So, we'll go to the app and we want to be logged in as an instructor and click on the Add Lesson button:


Home
Classes
Add Lesson
Lesson Number:
Lesson Title:
Lesson Body
Add Lesson
Welcome Back brad
My Classes
Logout