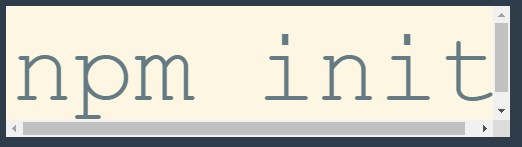
* Section slides: http://webdev.slides.com/coltsteele/mysql-105#/43
* In this section, we’ll move on to building our web application

# Introducing Express

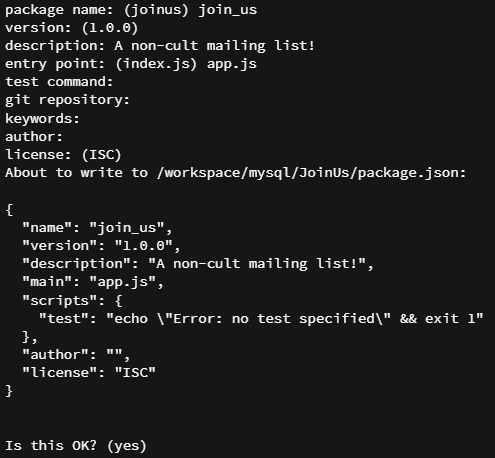
* **Express** is a web development framework for NodeJS that helps us make web applications faster
  + It essentially dictates how we write our code, but in return we can build web applications much faster than if we did it from scratch
  + It removes the “gunk” of web applications and lets us focus on the content that matters
  + Once we install it and include it in a file, it very quickly allows us to make a webpage
* Documentation: <https://expressjs.com/>
* Express is just one of many Node frameworks available

# NPM Init and package.json Files

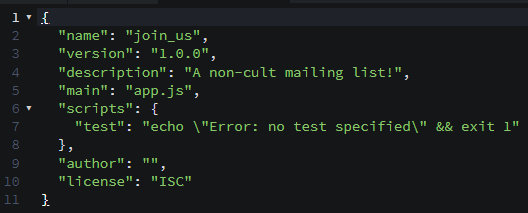
* Now we’ll get started on our barebones web application
* To install express, we run **npm install express**
* But another way is to use the following command



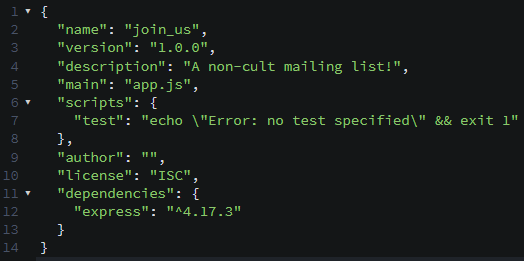
* + This creates a package.json file. Basically, the file acts as a log of all packages that we’ve installed. Then if you post your code on github, all someone has to do is run *npm install*, which will find the package.json file and then install all required package
* When we install Express using *npm init*, the package.json file accompanying Express will install all of the required packages. Very cool!
* Let’s create a new folder where we’ll build our web app, so that we don’t get things mixed up with our previous files that we’ve been using. We’ll call this folder **JoinUs**



* Running *npm init* and filling out the information generates a package.json file within our JoinUs folder. It’s now ready for us to install things

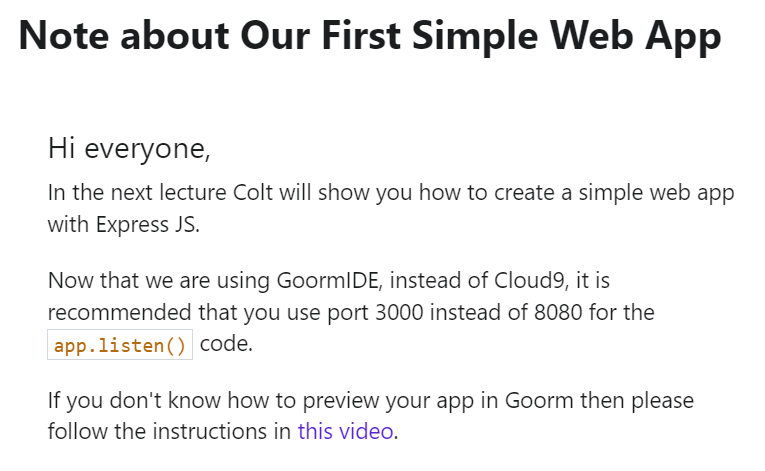


* Next, we install Express by running *npm install express --save*. The --save is important because it will save a record of the installation of Express into package.json



* We’ll need two more packages for our app: Faker and MySQL
  + *npm install faker mysql --save*
* Now that we have all of our packages installed, we just need to send the package.json file and our app code, and then they can install all the package on their own
  + All they need to run is *npm install* without any identifiers, and it will automatically install the packages in the nearest package.json file

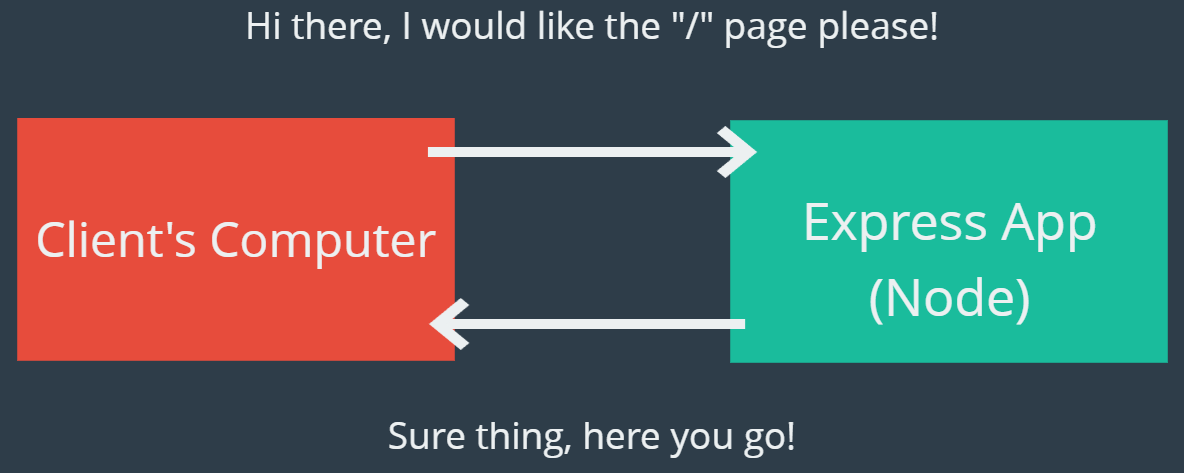
# A Note on Our Simple Web App



* <https://www.youtube.com/watch?v=6CYTQm-6t9w>

# Our First Simple Web App

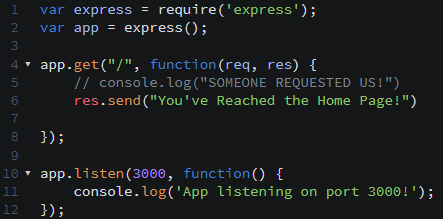
* In this video we write the code for an extremely simple web application
* Here’s the flow we’re trying to get:



* And here is the code from our super simple app:



* + The first two lines are the first steps to ANY express application
  + Per Ian’s suggestion above, we’ll change the port to 3000 instead of using 8080
  + The first block of code is known as the “route” – it does not execute unless an incoming **request** (req) is made to the path defined as the first argument in the app.get() clause
    - Once a request is received, we send a message held in the res.send() clause
    - So for every route, there is a callback that runs whenever we get a request
    - We want to run different code depending on what webpage is being requested
    - You can only have ONE res.send(), but that’s usually not an issue because most of the time you’ll be responding not with a message, but with a file, usually an HTML file or something of that sort
  + The second component is the app.listen() command that starts up the server. If you want, you can shoot a message to the console letting you know that the server is running
* Let’s make our app.js file:



* To summarize what we did:
  + We started by requiring Express and executed and saved it to the **app** variable
    - There’s a lot going on under the hood that we don’t need to worry about – that’s the point of a framework, it takes care of those details so that you don’t have to worry about them
  + Then we started up our server using **app.listen()**, which then sits and listens for a request
  + Then we write code the differentiates between different requests. In this case we requested a home page “/”. Otherwise, our code has no idea how to respond to other requests
    - There is a TON of information that comes in with the request, and you can look at it by printing the *req* variable to the console
* Instructor’s code

#### **CODE: Our First Simple Web App:**

Add to your app.js file:

var express = require('express');

var app = express();

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

res.send("HELLO FROM OUR WEB APP!");

});

app.listen(8080, function () {

console.log('App listening on port 8080!');

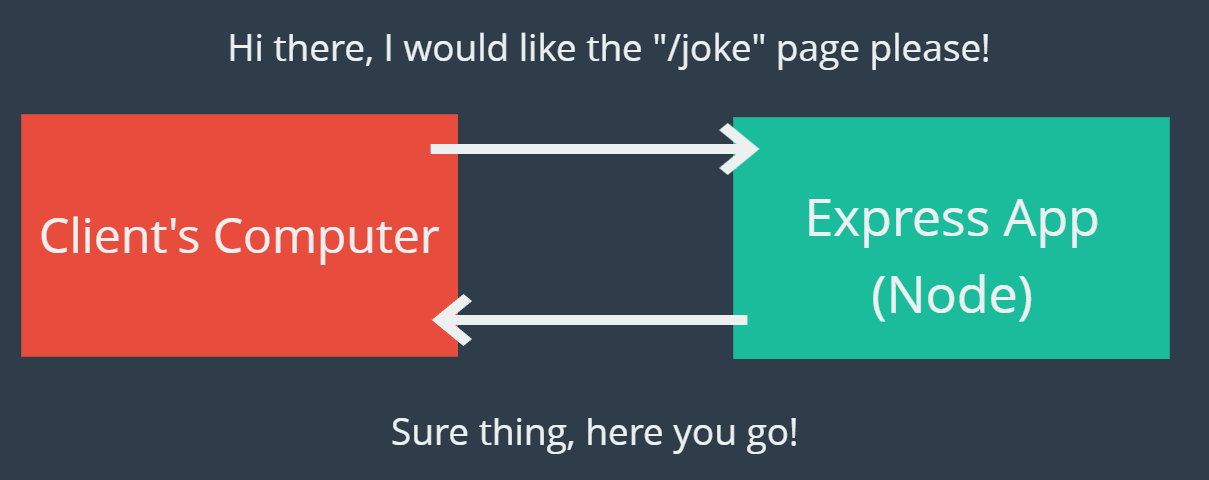
});

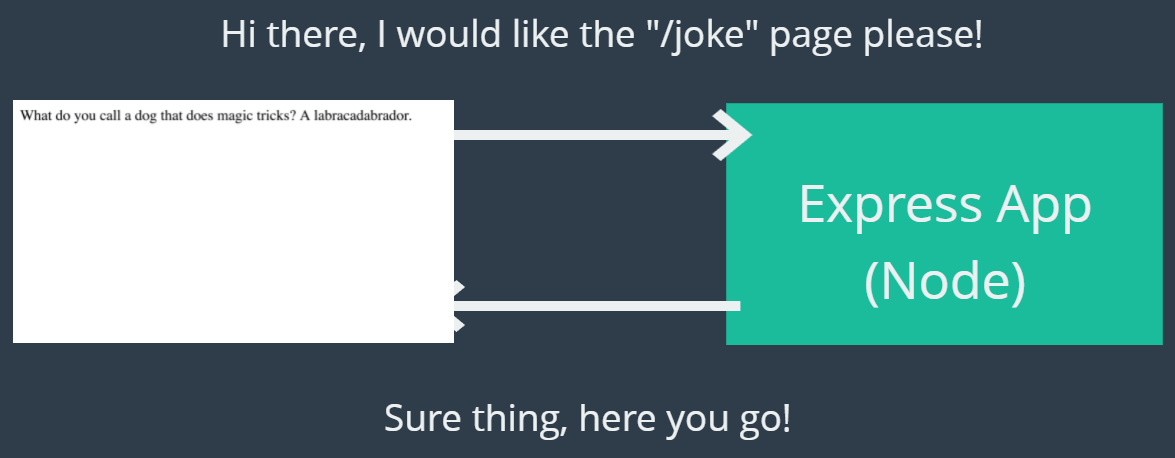
Remember to start the server up:

node app.js

# Adding Multiple Routes

* At this point our app can only respond to one type of request – the homepage, or the blank slash “/”. Any other type of request will error out
* The way to fix this is to add more routes!
* The first thing we’ll do is add a “/joke” page that gives a joke.

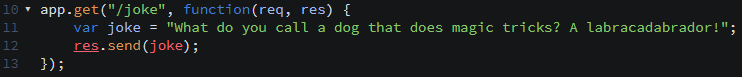




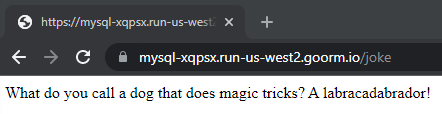
* + Here’s what our routing will do (pseudocode):



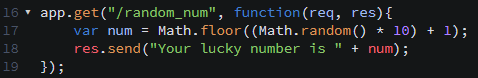
* + - Only one of these will run with each given request
  + Here’s the code – we have to add an app.get() call for the new routing:



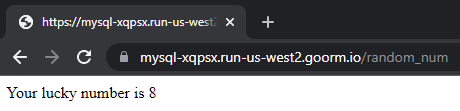
* + Now when we run the app and make a request to the joke route, the server will receive it and send the following response:



* Let’s wrap up with another route that returns a random number



* + The math.random() function can be used for this. This is an annoying function that generates a random number between 0 and 1, exclusive of 1. If you want a number between 1 and 10, you need to multiply the result by 10 and then add 1 to it.
    - The math.floor() trims off the decimals



* Instructor code:

#### **CODE: Adding Multiple Routes**

Add a /joke route:

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

var joke = "What do you call a dog that does magic tricks? A labracadabrador.";

res.send(joke);

});

Add a /random\_num route:

app.get("/random\_num", function(req, res){

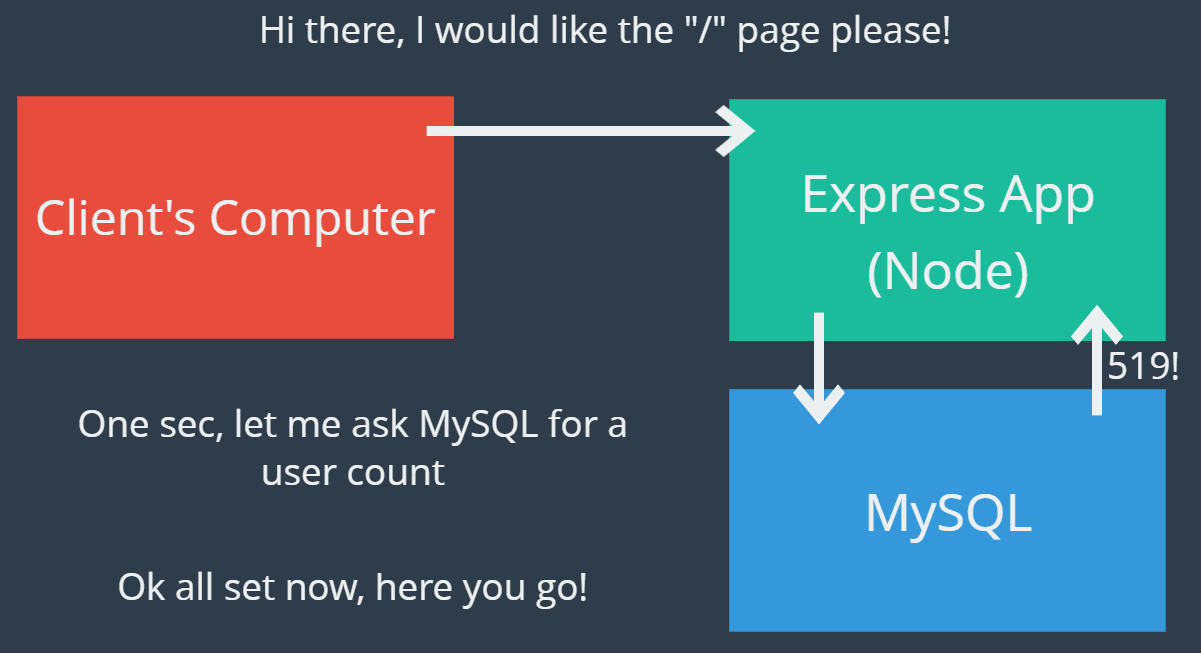
var num = Math.floor((Math.random() \* 10) + 1);

res.send("Your lucky number is " + num);

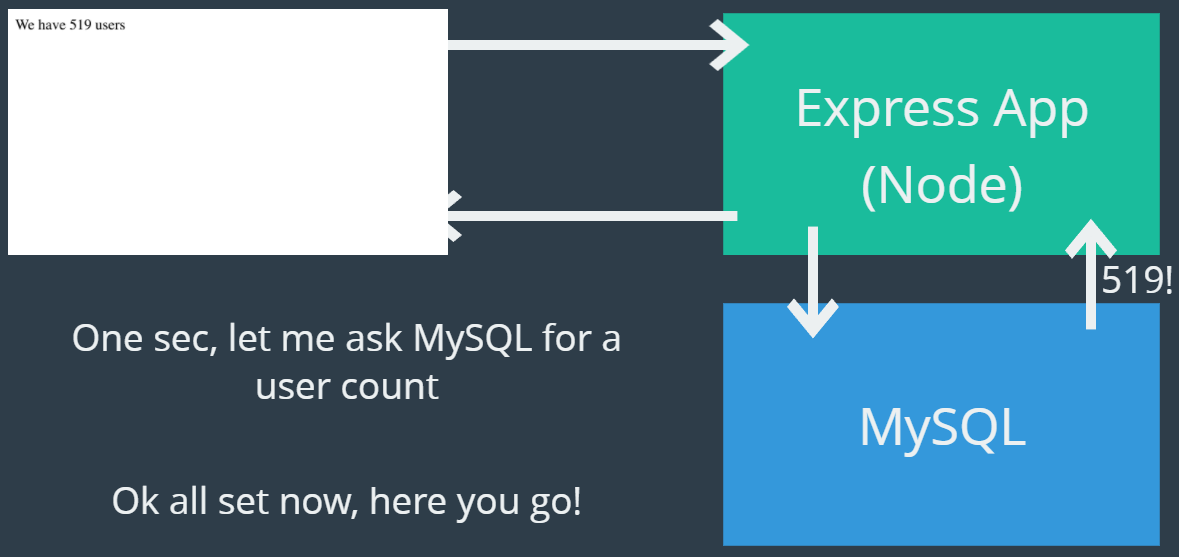
});

# Connecting Express to MySQL

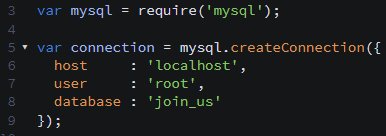
* We’ve previously created our database in MySQL with 500 users. Now we’re going to take our web application and display information about the number of users from the database.



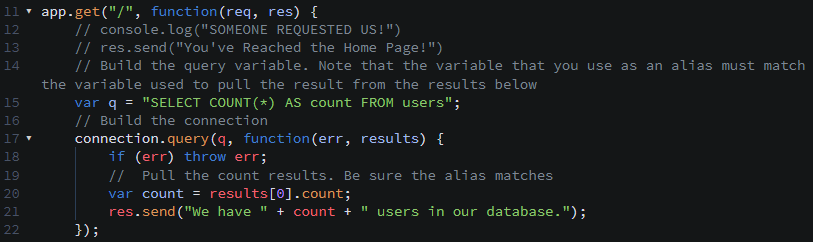
* + The user’s computer makes a request to the web app. The app recognizes that it needs to pull information from MySQL and does so. Then the app shoots the information back to the client’s computer

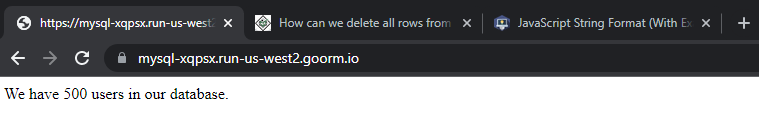


* First step: connect our app.js file to our MySQL database! We can do that by using the MySQL code from our previous dummy app.js file



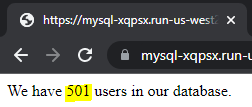
* Next, we update the home page response to find a count of number of users in the database





* To prove that it’s working, let’s add another user via MySQL and test it again:





* IMPORTANT NOTE: We don’t need to restart the app in order to see the database update. That would kind of defeat the purpose of having a database. Instead, the database will be on and updating all the time
* Next up, we have to create the form that allows people to input their email address and then execute an INSERT query to add themselves to our databases
* Instructor code:

#### **CODE: Connecting Express and MySQL**

Add the MySQL code inside of the root route:

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

var q = 'SELECT COUNT(\*) as count FROM users';

connection.query(q, function (error, results) {

if (error) throw error;

var msg = "We have " + results[0].count + " users";

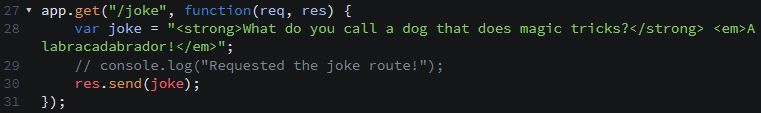
res.send(msg);

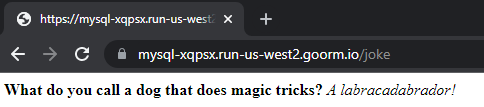
});

});

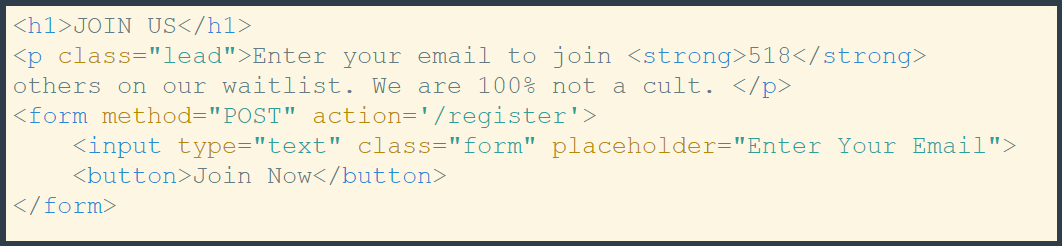
# Working with Views

* We’ll be using HTML and CSS to make our webpage actually look nice. We’ll be able to create a form and a submission button
* But first, we can actually put HTML inside of strings in JavaScript. Let’s take our dog joke example.
  + Here we’ll use the HTML tags <strong> and <em> for bold and italic text
  + This is a VERY BAD way to write HTML. You generally don’t want to write it in a single line, because it makes it difficult to change things and style them with CSS

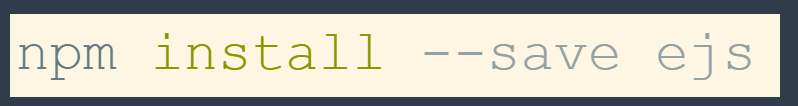




* Let’s start with an unstylized, boring but totally functional homepage. Here is all the HTML we need for our page



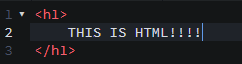
* Now, how do we shoot that back to the user? We use something called **EJS**
  + **EJS**, or embedded JavaScript, is a *templating language*
  + It’s an alternative to writing plain HTML and allows us to write variable values and code logic.
    - For instance, our count of number of users is a variable. You cannot use variables in HTML, so EJS will allow us to embed JavaScript (which CAN have variables) into HTML
  + Let’s install it



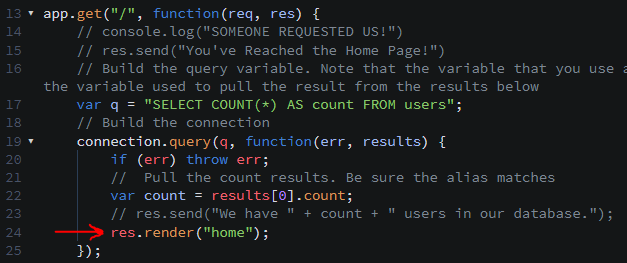
* Now we need to configure our Express application with the following code:
  + app.set("view engine", "ejs");
  + **app.set()** is a method that allows us to set particular settings in the app. “View engine” is a common one, but there are others as well
* How do we actually render these files to serve to the user? We start with **res.render(‘home’);** which will render
  + Behind the scenes, express will search for a Views directory and within there will search for a file called “home.ejs”

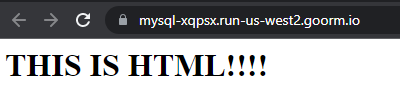


* + So we create a folder called “views” and a filed called “home.ejs”, where we write our HTML code

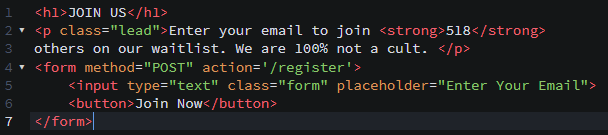


* + Now we go into our **app.js** file and within the app.get() function, we send a res.render() instead of a res.send()
  + Let’s test her out!

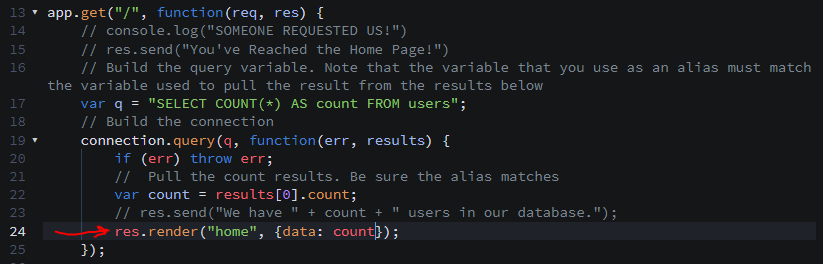




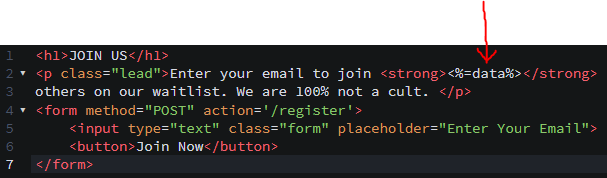
* Let’s now use the actual HTML we need:

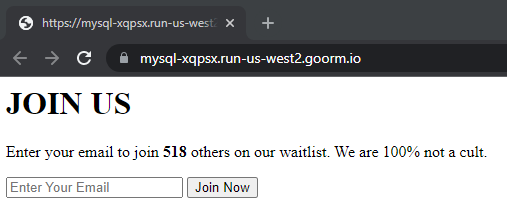


* + 518 is hard-coded for now, we’ll be changing that
* Now, how do we get our dynamic count of users into this HTML? What we do is pass in a comma in res.render(), and we pass in a JavaScript object where we can pass data through
  + We will pass in {data: count}. This is distinct from the variable **count**. We’ll change it later
  + What this does is says *take the variable* ***count****, send it to the HTML template, and give it the name of* ***data***, which serves as the label or key for our data that we’re passing in
    - We can pass in any additional data if we have it, we are not limited to just *count*. And any data that we pass will be accessible within “home.ejs”



* + But we need a way of identifying *data* as something that is NOT HTML, but a variable. Here’s how we do it:





* To summarize:
  + The app.get(“/”) activates the route when you go to the URL
  + When you hit the route, the app.get() code runs and figures out how many users are in our MySQL database by doing a query. This value is stored as the variable **count** because that’s what we aliased it as with our query
  + Then, we render the “home” HTML file, which contains a variable containing the value that was retrieved by the app.get() code
* Instructor code:

<h1>JOIN US</h1>

<p class="lead">Enter your email to join <strong><%= count %></strong>

others on our waitlist. We are 100% not a cult. </p>

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

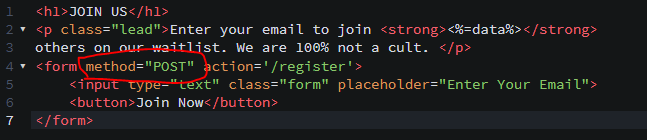
<input type="text" class="form" name="email" placeholder="Enter Your Email">

<button>Join Now</button>

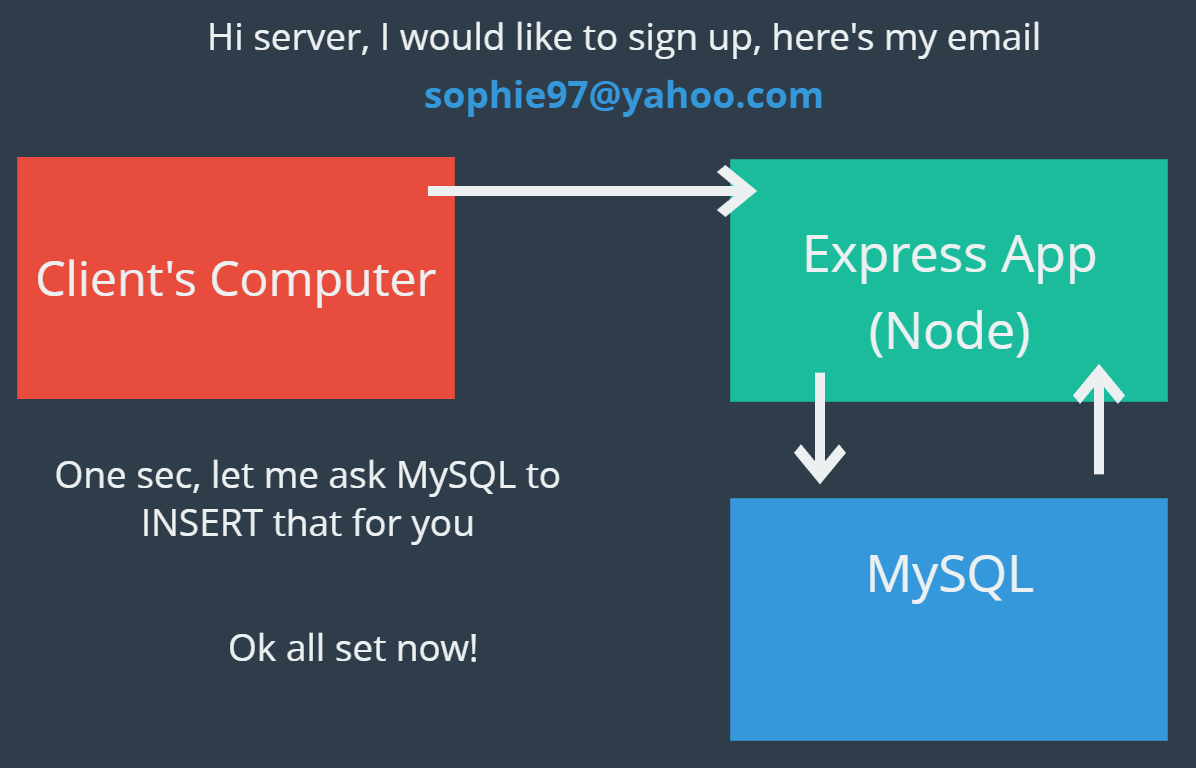
</form>

# Connecting the Form

* So we’ve got our barebones webpage up and running. Now we need to make that email entry form interactive
  + We need to define what happens when someone clicks the “Join Now” button, and define a route for that event
  + Another problem is getting the data from the form to the destination. We can do this using a **post request**, which allows us to send data with our request
* We’ve actually already done this in our code. Below, we define the form as having a POST method and an action to take us to the /register page, which does not yet exist



* Now we need to define a post route at /register. Again, the reason we’re using POST is because it allows us to send the form data
  + Other requests can be submitted, such as a GET request
* We also need to give our data a **name** to be sent under. This allows us to send our data, and the name will be appended to the URL that we request
* So, how do we create a route that will listen to our post request?
  + <http://webdev.slides.com/coltsteele/mysql-105#/72/0/7>



* + We use the **app.post()** method within our app! This code will only be triggered when we sent a POST request to /register. It will not trigger if we send a GET request.



* But now we need to do stuff with the data we’re sending (i.e. the text we’re entering into the form”
  + Install body parser
  + Require it within our app
  + Tell our app to use body parser

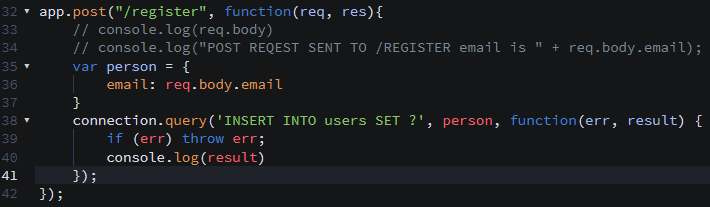


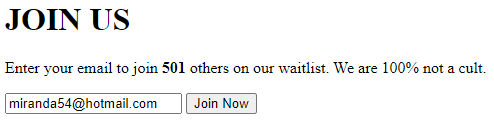


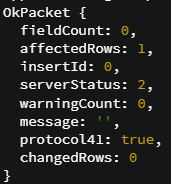
* + Body parser did a lot of the annoying drill-down work for us. What does the body look like?

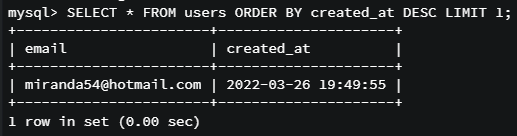


* We’ve successfully isolated the email address entered into the form, and we can save it as a variable within app.js. Now we need to perform a query on the MySQL server to insert that email address into it

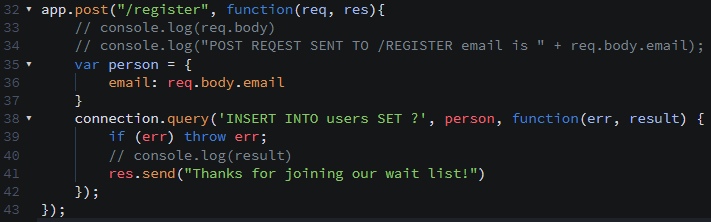


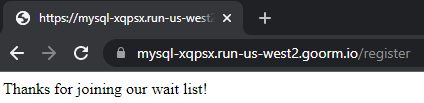




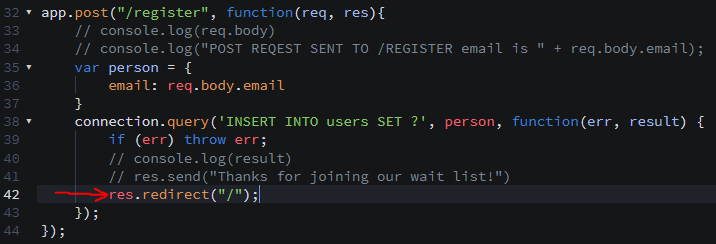


* Now what? After the email is added, what do we do? One option is to send them a nice message





* Another option is to use res.redirect() to punt the user back to the homepage



* Instructor code

The '/register' post route:

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

var person = {email: req.body.email};

connection.query('INSERT INTO users SET ?', person, function(err, result) {

console.log(err);

console.log(result);

res.redirect("/");

});

});

# Styling Part 1

* Now it’s time to turn our ugly looking website and style it using CSS
* The way it usually works is that you create an external style sheet and then connect it to HTML elements! The connection bit happens with this line



* + This of course assumes that our CSS file lives in the /public folder