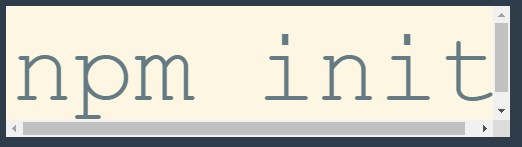
* Section slides: http://webdev.slides.com/coltsteele/mysql-105#/43
* In this section, we’ll move on to building our web application

# Introducing Express

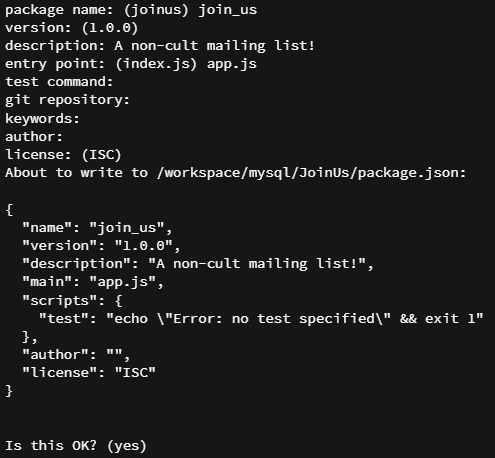
* **Express** is a web development framework for NodeJS that helps us make web applications faster
  + It essentially dictates how we write our code, but in return we can build web applications much faster than if we did it from scratch
  + It removes the “gunk” of web applications and lets us focus 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

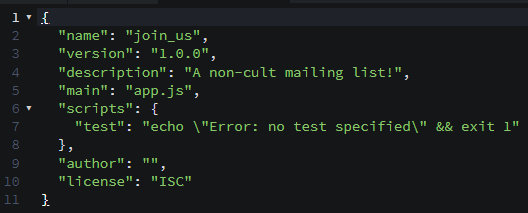
* Now we’ll get started on our barebones web application
* To install express, we run **npm install express**
* But another way is to use the following command



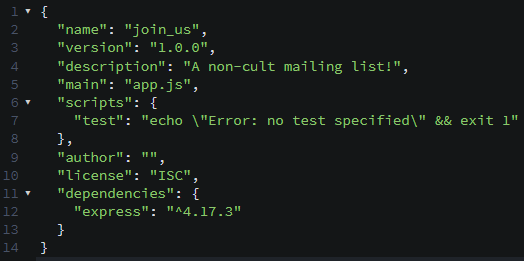
* + This creates a package.json file. Basically, the file acts as a log of all packages that we’ve installed. Then if you post your code on github, all someone has to do is run *npm install*, which will find the package.json file and then install all required package
* When we install Express using *npm 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

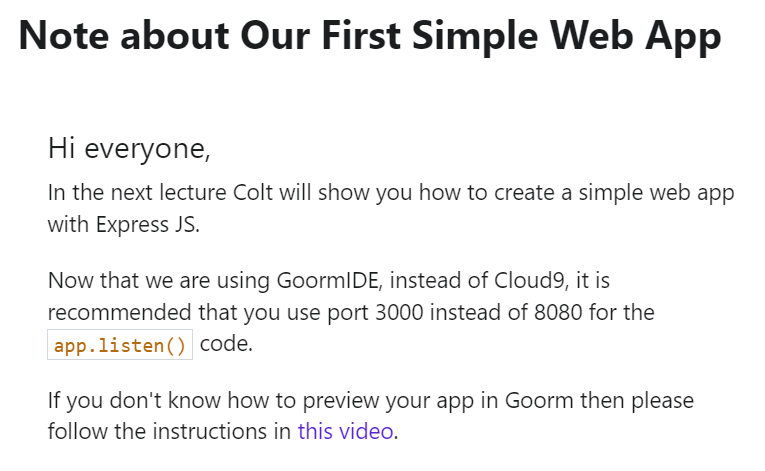


* Next, we install Express by running *npm install express --save*. The --save is important because it will save a record of the installation of Express into package.json



* 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

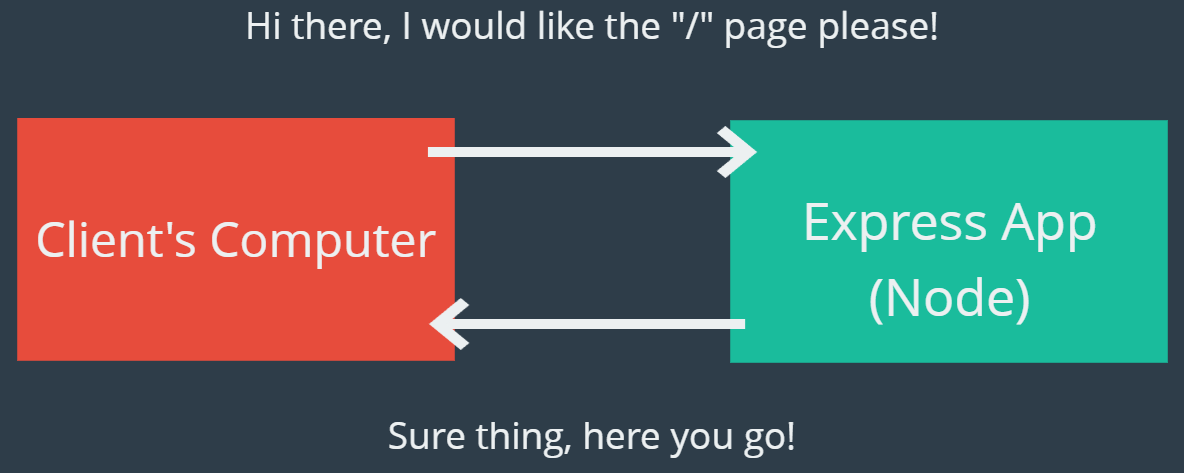
# A Note on Our Simple Web App



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

# Our First Simple Web App

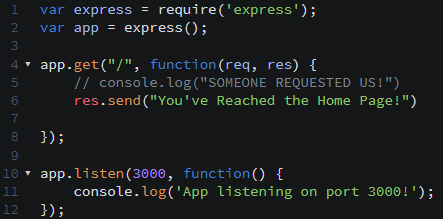
* In this video we write the code for an extremely simple web application
* Here’s the flow we’re trying to get:



* 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!');

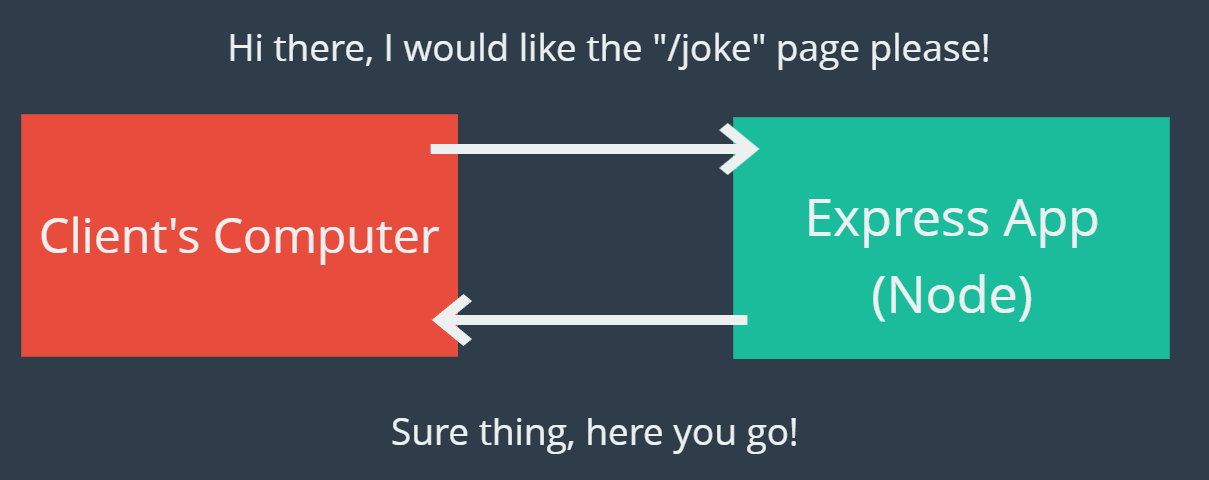
});

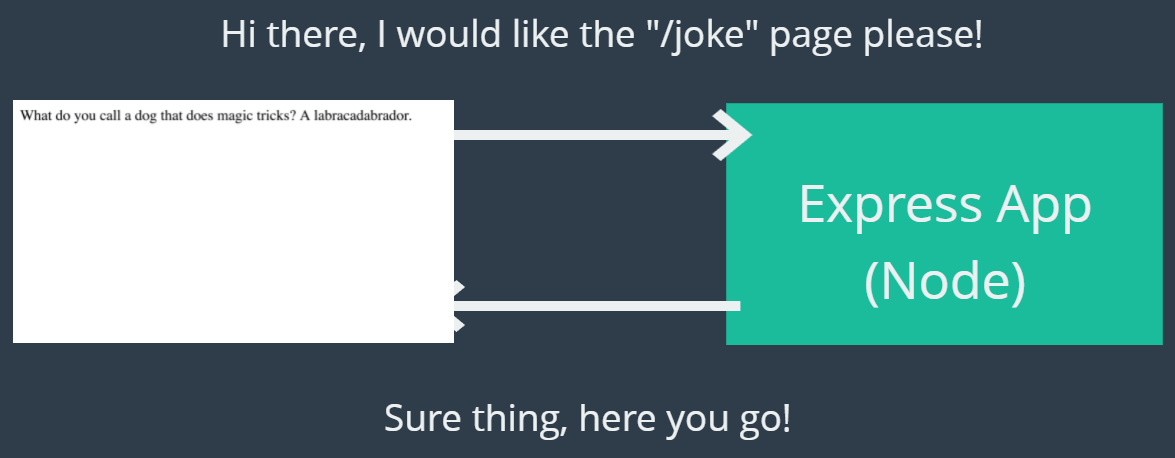
Remember to start the server up:

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.

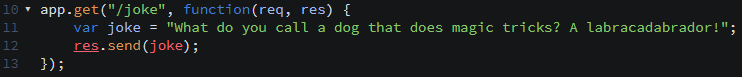




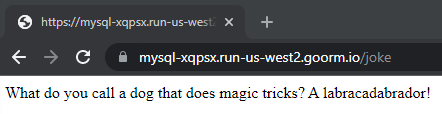
* + Here’s what our routing will do (pseudocode):



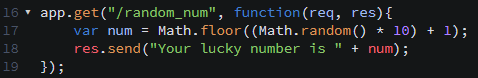
* + - Only one of these will run with each given request
  + Here’s the code – we have to add an app.get() call for the new routing:



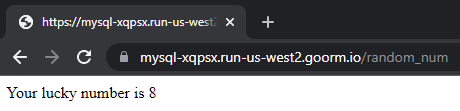
* + Now when we run the app and make a request to the joke route, the server will receive it and send the following response:



* 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){

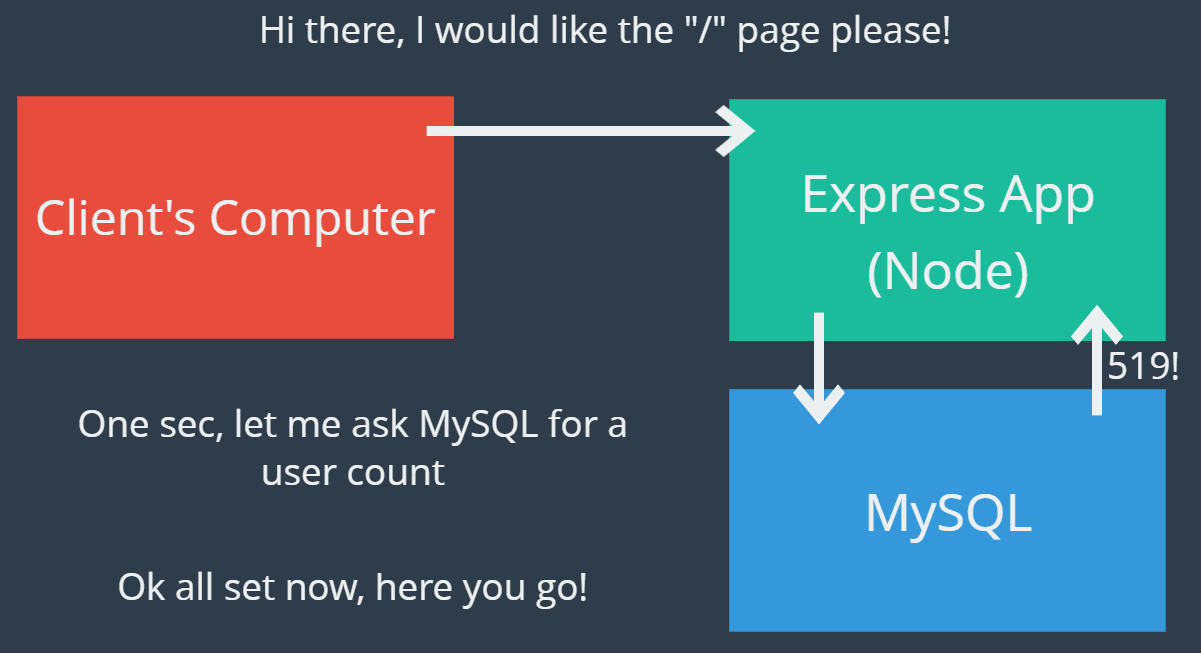
var num = Math.floor((Math.random() \* 10) + 1);

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

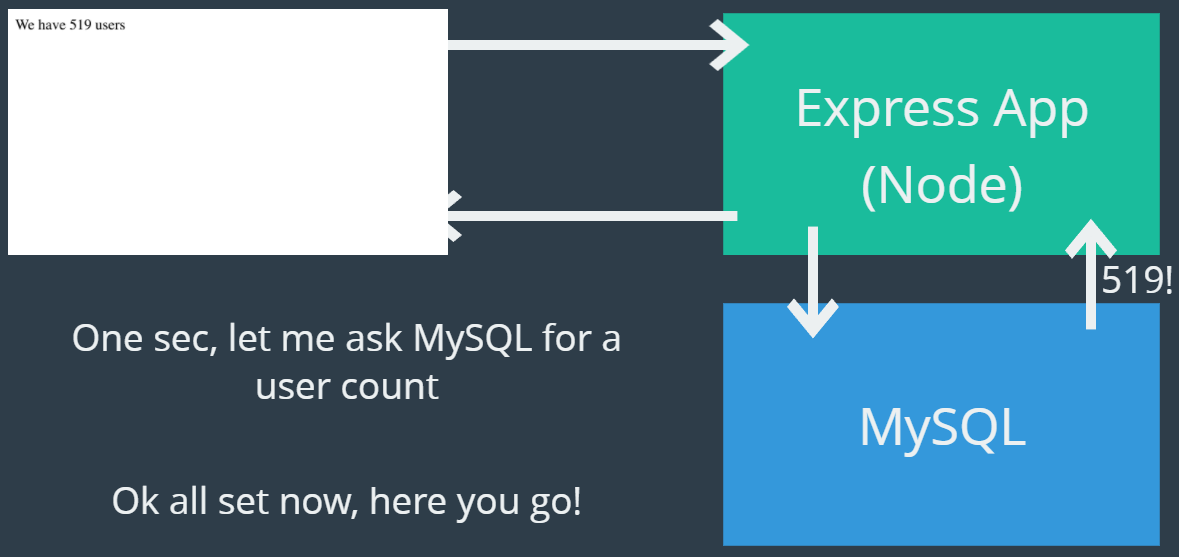
});

# Connecting Express to MySQL

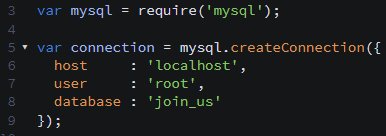
* We’ve previously created our database in MySQL with 500 users. Now we’re going to take our web application and display information about the number of users from the database.



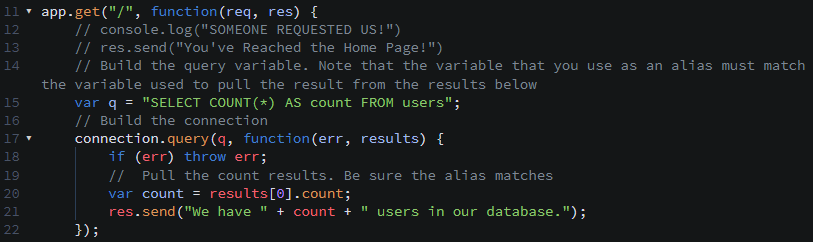
* + The user’s computer makes a request to the web app. The app recognizes that it needs to pull information from MySQL and does so. Then the app shoots the information back to the client’s computer

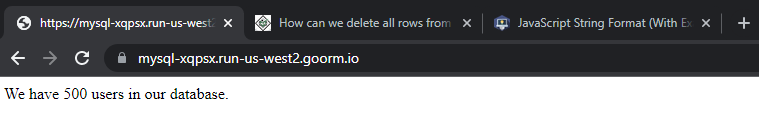


* First step: connect our app.js file to our MySQL database! We can do that by using the MySQL code from our previous dummy app.js file



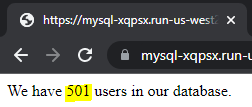
* Next, we update the home page response to find a count of number of users in the database





* 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);

});

});