The Dead-Simple Step-By-Step Guide for Front-End Developers to Getting Up and Running With Node.JS, Express, and MongoDB

Set up the full stack and have a webpage running in 30 minutes. Make it talk to your DB in another 30.

By Christopher Buecheler | @closebracejs | Last Updated: 1st Feb 2019
Posted In: Node.js Basics

This tutorial has been tested with Node.js v10.15.x, MongoDB v4.0.x, and Express v4.16.x. It will not work properly with older versions on some systems.

You can find/fork the entire sample project on GitHub

This tutorial was originally published on cwbuecheler.com and does not include a video component.

Updated February 1, 2019!

Introduction

There are approximately one hundred million tutorials on the web for getting a "Hello, World!" app running with Node.js. This is great! It's especially great if your goal is to greet the world and then give up on your web career and go spend the rest of your life as, like, a jockey or something. That doesn't really describe most of us, so we go looking for more tutorials.

In my experience, the "next level" tutorials out there seem about 30 levels further along. We go from "Hello, World!" to building out an entire blogging system with comments. Which is also great, but a lot of times those tutorials assume the reader has done a whole bunch of intermediate fiddling, and they often drop a bunch of big functions on you all at once. I tend to learn best by making lots of smaller, intermediate steps, and I don't think I'm the only one.

I'm not the only one, right?

Well, good news, everyone! I've done the fiddling and read a bunch of tutorials and shouted at my command prompt until things finally worked. I have a web project up and running which uses Node.JS, the Express framework, the EJS HTML pre-processor, and MongoDB for data. I can read to and write from the DB. From there, the sky's the limit.

Here's the deal: I'm going to show you how to get all of this stuff set up. I'll be assuming that you're a front-end developer who knows HTML5/CSS3/JavaScript well enough that I don't have to explain those. If that's you, then this should be a solid primer.

Your app will look pretty, it will connect to a DB, it'll get some results, and it'll do stuff with those results. Then for kicks we'll also make it save data to the DB. Through it all, I will explain what the code does, and how to write it, instead of just giving you massive functions to stare at. We'll go from nothing even installed, to a DB-driven web app written in a language you fully understand, and the foundation necessary to build additional functionality into your app. And we'll do it in about 60 minutes of installation and coding time. Is that awesome? I submit that it is.

Let's go.

Part I – 15 minutes of installing

If you're really starting from scratch, then getting everything up and running takes a little bit of time. None of it is difficult. I run Windows 10 on my main machine, so it'll be slightly different for those on a Mac or Ubuntu or other *nix system, but it's principally the same thing in all cases.

Step 1 – Install Node.js

This is really easy. Hit the Node.js website and click the big green Install button for the LTS (long-term-stable) version. It'll detect your OS and give you the appropriate installer. Run the installer. That's it, you have installed Node.js and, equally important,

NPM, which lets you add all kinds of great stuff to Node quickly and easily.

- Open a command prompt
- cd to the directory in which you wish to keep your test apps (for the purposes of this tutorial, C:\node).

Step 2 – Install Express Generator

Now that we have Node running, we need the rest of the stuff we're going to actually use to create a working website. To do that we're going to install Express, which is a framework that takes Node from a barebones application and turns it into something that behaves more like the web servers we're all used to working with (and actually quite a bit more than that). We need to start with Express-Generator, which is actually different than Express itself ... it's a scaffolding app that creates a skeleton for express-driven sites. In your command prompt, type the following:

Command C:\node>

```
C:\node>npm install -g express-generator
```

The generator should auto-install, and since it (like all packages installed with -g) lives in your master NPM installation directory, it should already be available in your system path. So let's use our generator to create the scaffolding for a website.

Step 3 – Create an Express Project

We're going to use Express and EJS, but not the Stylus CSS preprocessor (which people sometimes use in this stack). We're just going to use straight CSS for right now. We have to use EJS or another templating engine to gain access to our Node/Express-based data. EJS's not hard to learn if you already know HTML, since it's just HTML with some embedded JavaScript. Express defaults to Jade, but we're not using that because a) it's deprecated in favor of the new version, which is called Pug, and b) it's indentation-based and I don't like indentation-based markup. It's too easy for one accidental extra tab or space to break your whole page.

A quick note on indentation: everything is this tutorial has been normalized to 2-space indents because that's become standard in the JavaScript world. If you want to use four spaces, or actual tabs, that's just fine by me.

Anyway, still in c:\node or wherever you're storing your node apps, type this:

Command C:\node\

```
C:\node>express --view="ejs" nodetest1
```

Hit enter and watch it go. You'll see something like this:

Command C:\node\

```
C:\node>express --view="ejs" nodetest1

   create : nodetest1\
   create : nodetest1\public\
   create : nodetest1\public\javascripts\
   create : nodetest1\public\images\
   create : nodetest1\public\stylesheets\
   create : nodetest1\public\stylesheets\style.css
   create : nodetest1\routes\
   create : nodetest1\routes\index.js
   create : nodetest1\routes\users.js
   create : nodetest1\views\
   create : nodetest1\views\error.ejs
   create : nodetest1\views\index.ejs
   create : nodetest1\app.js
   create : nodetest1\package.json
   create : nodetest1\bin\
   create : nodetest1\bin\www

   change directory:
     > cd nodetest1

   install dependencies:
     > npm install

   run the app:
     > SET DEBUG=nodetest1:* & npm start
```

Step 4 – Edit Dependencies

MAKE SURE TO CD INTO YOUR `nodetest` FOLDER. OK, now we have some basic structure in there, but we're not quite done. You'll note that the express-generator routine created a file called `package.json` in your `nodetest1` directory. Open this up in a text editor and it'll look like this:

C:\node\nodetest1\package.json

```
{
  "name": "nodetest1",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "cookie-parser": "~1.4.3",
    "debug": "~2.6.9",
    "ejs": "~2.5.7",
    "express": "~4.16.0",
    "http-errors": "~1.6.2",
    "morgan": "~1.9.0"
  }
}
```

This is a basic JSON file describing our app and its dependencies. We need to add a few things to it. Specifically, calls for MongoDB and Monk. We could do that by editing our file ... but we can also install those dependencies with NPM and it'll add them automatically. Head back to your command line and type the following:

```
Command C:\node\nodetest1

C:\node\nodetest1>npm install --save monk@^6.0.6 mongodb@^3.1.13
```

This will add those two dependencies to our `package.json` file (you can switch back to your text editor to check). It'll also install them for use in our app ... but right now those are the *only* two dependencies we've installed, because there's an important step when getting set up with Express that we haven't done yet.

Step 5 – Install Dependencies

Now we've defined our dependencies and we're ready to go. Note that new versions of NPM modules are frequently rolled out. The versions listed in this tutorial are proven to work; if you go with the latest versions, I'm afraid you're on your own.

Return to your command prompt and type this:

```
Command C:\node\nodetest1\

C:\node\nodetest1>npm install
```

That will read the JSON file and install all the stuff listed in the dependencies object that aren't yet installed (yes, including Express – we installed the generator globally, but we still have to install the actual module inside this one particular project). You might get a few vulnerability warnings depending on versions, but you can ignore them. We're not deploying this anywhere, so security's not really a concern. Anyway, at the time of the "last updated" date above, there are zero known vulnerabilities in the project.

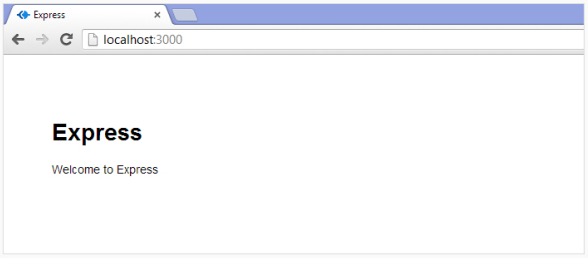
Once NPM has run its course, you should have a `node_modules` directory which contains all of our dependencies for this tutorial. This will also create a file called `package-lock.json` which you can ignore completely. It's important, but only if you're an experienced Node developer building production-level apps. You now have a fully-functioning app ready and waiting to run, so let's test out our web server! Type the following:

```
Command C:\node\nodtest1\

C:\node\nodetest1>npm start
```

Hit enter, and you'll get this:

```
Node Console

> nodetest1@0.0.0 start C:\node\nodetest1
> node ./bin/www
```

Everything working? Awesome! Open a browser and head for http://localhost:3000 where you will see a welcome to Express page.



You are now running your own Node JS webserver, with the Express engine and EJS HTML preprocessor installed. Not so tough, huh?

Part 2 – OK, Fine, let's do "Hello, World!"

Fire up your favorite text editor or IDE. I like and use both Sublime Text and VS Code. Point it at your `nodetest1` directory and open `app.js`. This is kind of the heart of your, well, app. Not a big surprise there. Here's a breakdown of what you're going to see:

```
C:\node\nodetest1\app.js

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

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

This creates a bunch of basic JavaScript variables and ties them to certain packages, dependencies, node functionality, and routes. Routes are kind of like controllers in this setup – they direct traffic and also contain some programming logic (you can establish a more traditional MVC architecture with Express if you like. That's outside of the scope of this article). Back when we set up this project, Express created all of this stuff for us. We're going to totally ignore the user route for now and just work in the top level route (controlled by `c:\node\nodetest1\routes\index.js`).

C:\node\nodetest1\app.js

```
var app = express();
```

This one's important. It instantiates Express and assigns our app variable to it. The next section uses this variable to configure a bunch of Express stuff.

C:\node\nodetest1\app.js

```
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

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')));

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

This tells the app where to find its views, what engine to use to render those views (EJS), and calls a few methods to get things up and running. Note also that the final line of the second block is telling Express to serve static objects from the /public/ dir, but to make them actually seem like they're coming from the top level (it also does this with the views directory). For example, the images directory is `C:\node\nodetest1\public\images` ... but it is accessed at `http://localhost:3000/images`.

That final block, the last two lines, is important. It's telling Express that requests to `http://localhost:3000/` should use the index router (which is defined up near the top of the file), and requests to `http://localhost:3000/users` shold use the users router file.

C:\node\nodetest1\app.js

```
// 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');
});
```

These are error handlers for development and production (and 404's). We're not really worrying about the different between those two right now, but basically if your app is in development mode, your errors will give you more information. Obviously you don't want to print a stack trace out on a production site that anyone on the web can see.

C:\node\nodetest1\app.js

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

A core part of Node is that basically all modules export an object which can easily be called elsewhere in the code. Our master app exports its app object.

Now then, let's make stuff. We're not going to just stick "Hello, World!" on our index page. Instead we're going to use this as an opportunity to learn a bit more about routes and to take a look at how EJS works for putting pages together.

We're going to start by adding a new `app.use` directive to `app.js`. Find the section that looks like this:

C:\node\nodetest1\app.js

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

These directives are telling Express what route files to use. Now, normally I'd advocate setting up separate route files for different parts of your app. For example, the users route file might contain routes for adding users, deleting them, updating them, and so forth, while a new route file called "locations" might handle adding, editing, deleting and displaying location data (in an app for which that was required). In this case, to keep things simple, we're going to do everything in the index router. That means you can completely ignore the /users line.

Remember that the Express scaffolding already defined the `routes` variable and pointed it at the index router. We're going to add a `helloworld` method to that

pointed it at the index router. We're going to add a `helloworld` method to that router which will render a different page than the default. In your text editor, open up your `routes` folder, find `index.js`, and open it. It will look like this:

C:\node\nodetest1\routes\index.js

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

/* GET home page. */
router.get('/', function(req, res) {
    res.render('index', { title: 'Express' });
});

module.exports = router;
```

Pretty sparse, right? Basically we're requiring our Express functionality, then attaching a "router" variable to Express's router method, then using that method when an attempt is made to HTTP get the top level directory of our website. Finally we export our router function back to our app.

We can easily clone that get function for another page, so let's do that. At the bottom of the file, just above the module.exports line, add this code:

C:\node\nodetest1\routes\index.js

```
/* GET Hello World page. */
router.get('/helloworld', function(req, res) {
    res.render('helloworld', { title: 'Hello, World!' });
});
```

That's all it takes to handle routing the URI, but we don't have any actual page for res.render to ... render. That's where EJS comes in. Open up your views folder, and then go ahead and open `index.ejs`. Before you do anything else, save it as **`helloworld.ejs`**.

Now take a look at the code:

C:\node\nodetest1\views\helloworld.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

This is pretty straightforward. Note the use of the "title" variable which we set above, in our `index.js` route. This means we don't even have to change the text at all in order for it to show different stuff from the home page. But let's change it anyway to:

```
<p>Hello, World! Welcome to <%= title %></p>
```

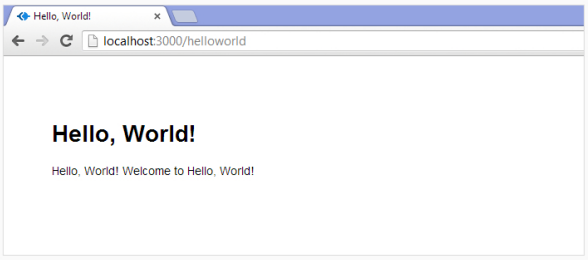
Save the file, go to your command prompt, `ctrl-c` (or `cmd-c`) to kill your server if it's already running, and then type:

Command C:\node\nodetest1\

```
npm start
```

In order to restart the server. By the way, this seems a good time to mention: changes to EJS templates do not require a server restart, but basically whenever you change a js file, such as `app.js` or the route files, you'll need to restart to see changes.

SO ... with the server restarted, navigate to `http://localhost:3000/helloworld` and enjoy the completely asinine text that gets displayed:



OK! So now we've got our router routing us to our view, which we are viewing. Let's do some modeling. I'll give you a moment if you need to fix your hair or makeup.

Part 3 – Create our DB and read stuff from it

Step 1 – Install MongoDB

We're breaking this up into two parts: Windows and Mac. We'll do Windows first.

Windows

We're leaving our text editor for a bit and going back to our command prompt. Well, first we're going to our web browser, pointing it to http://mongodb.com/ and downloading Mongo. You don't want the "try free" button, which is going to push their enterprise server or cloud server on you, neither of which you need for local

development. Instead, hover over "products" and click "MongoDB Server" - this will take you to the Community Server download page. It will auto-detect your OS and give you the appropriate download link.

Click the download button and run the installer. If you're on Windows, it'll offer to install a GUI called Compass. You can do that if you want, but we won't be using it in this tutorial. It'll also ask if you want to register MongoDB as a service. The answer is "yes" because it makes everything else a lot easier. The DB server will just run in the background whenever your machine is on (using minimal resources, don't worry) and you won't need to start it up every single time.

Once it's installed, you're all set.

Mac

If you're on OSX, it's a little more complicated to set MongoDB up as a background service, but it can be done. I recommend using Homebrew to do the installation. If you're not familiar with Homebrew, that's OK. It comes with OSX and you don't really need to do anything to configure it. Head for your terminal and type the following:

Terminal ~\

```
brew update
```

This will snag all the latest info Homebrew needs to make sure it's installing the right stuff. Now let's install MongoDB with this command:

Terminal ~\

```
brew install mongodb
```

Let that run and it'll install what you need. Next step, make a directory in which to store your data. MongoDB themselves suggest `/data/db` because that's what MongoDB looks for by default. I'd go with that, too. Here's what to type:

Terminal ~\

```
mkdir -p /data/db
```

Because you used Homebrew to install MongoDB, you can pretty easily start MongoDB up as a service by typing:

Terminal ~\

```
brew services start mongodb
```

Once it's installed and started, we can move on to working with it!

Step 2 – Run mongo

Now open a new command prompt or terminal window because we're going to be talking to this database server while also running our Express server. In that new window, we're going to use a utility that comes with MongoDB, which is called "Mongo". It's basically a way to access your DB from the command line. Just type this:

Terminal ~\ or C:\Node

```
mongo
```

You'll see something like the following:

Mongo Console

```
MongoDB shell version v4.0.5
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 4.0.5
Server has startup warnings:
[a bunch of warnings that you can ignore for this tutorial]
[Possibly an annoying ad for their cloud service]
>
```

All right, you've got MongoDB up and running, and you've connected to it with the client. We'll use this client to manually work on our database, for a bit, but it's not necessary for running the website. Only the server daemon (mongod) is needed for that.

Step 3 – Create a Database

The first thing we're going to do is create a database in which to store stuff. In your Mongo console, type the following:

Mongo Console

```
use nodetest1
```

Now we're using the database "nodetest1," which is great except nothing actually exists yet. To make the database exist, we have to add some data. We're going to start off by doing that right inside of the Mongo client.

Step 4 – Add some Data

My favorite thing about MongoDB is that it uses JSON for its structure, which means it was instantly familiar for me. If you're not familiar with JSON, you'll need to do some reading, as I'm afraid that's outside the scope of this tutorial.

Let's add a record to our collection. For the purposes of this tutorial, we're just going to have a simple database of usernames and email addresses. Our data format will thus look like this:

Mongo Console

```
{
    "_id" : 1234,
    "username" : "cwbuecheler",
    "email" : "cwbuecheler@nospam.com"
}
```

You can create your own `_id` assignment if you really want, but I find it's best to let Mongo just do its thing. It will provide a unique identifier for every single top-level collection entry. Let's add one and see how it works. In your Mongo client, type this:

Mongo Console

```
db.usercollection.insert({ "username" : "testuser1", "email" : "testuser1@testdomain.com" })
```

Something important to note here: that `db` stands for our database, which as mentioned above we've defined as `nodetest1`. The `usercollection` part is our collection. Note that there wasn't a step where we created the `usercollection` collection. That's because the first time we add to it, it's going to be auto-created. Handy. OK, Hit enter. Assuming everything went right, you should see:

Mongo Console

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

which is Mongo's odd way of telling you everything went according to plan. That's not very exciting, though, so type this:

Mongo Console

```
db.usercollection.find().pretty()
```

In case you're curious, the .pretty() method gives us linebreaks. It will return:

Mongo Console

```
{
    "_id" : ObjectId("5202b481d2184d390cbf6eca"),
    "username" : "testuser1",
    "email" : "testuser1@testdomain.com"
}
```

Except, of course, your `ObjectId` will be different, since as mentioned, Mongo is automatically generating those. That's all there is to writing to MongoDB from the client app, and if you've ever worked with JSON services before, you are probably going "oh, wow, that's going to be easy to implement on the web." ... you're right!

A quick note on DB structure: obviously in the long run you're unlikely to be storing everything at the top level. There are a ton of resources on the internet for schema design in MongoDB. For example, our own Five Minute React series of tutorials dives into it.

Now that we've got one record, let's add a a couple more. In your Mongo console, type the following:

Mongo Console

```
newstuff = [{ "username" : "testuser2", "email" : "testuser2@testdomain.com" }, { "username" : "te
db.usercollection.insert(newstuff);
```

Note that, yes, we can pass an array with multiple objects to our collection. Handy! Another use of db.usercollection.find().pretty() will show all three records:

Mongo Console

```
{
        "_id" : ObjectId("5202b481d2184d390cbf6eca"),
        "username" : "testuser1",
        "email" : "testuser1@testdomain.com"
}
{
        "_id" : ObjectId("5202b49ad2184d390cbf6ecb"),
        "username" : "testuser2",
        "email" : "testuser2@testdomain.com"
}
{
        "_id" : ObjectId("5202b49ad2184d390cbf6ecc"),
        "username" : "testuser3",
        "email" : "testuser3@testdomain.com"
}
```

Now we're going to start actually interacting with the web server and site that we set up earlier. You can quit out of the MongoDB console with ctrl-c or by typing "exit" and hitting enter. We're done here.

Step 5 – Hook Mongo up to Node

This is where the rubber meets the road. Let's start by building a page that just spits out our DB entries in a mildly pretty form. Here's the HTML we're shooting to generate:

```
<ul>
    <li><a href="mailto:testuser1@testdomain.com">testuser1</a></li>
    <li><a href="mailto:testuser2@testdomain.com">testuser2</a></li>
    <li><a href="mailto:testuser3@testdomain.com">testuser3</a></li>
</ul>
```

I know this isn't rocket science, but that's the point. We're just doing a simple DB read-and-write in this tutorial, not trying to build a whole website. First things first, we need to add a few lines to our main `app.js` file—the heart and soul of our app—in order to actually connect to our MongoDB instance. Open `C:\node\nodetest1\app.js` and at the top you'll see:

C:\node\nodetest1\app.js

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

Now add these three lines:

C:\node\nodetest1\app.js

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

// New Code
var monk = require('monk');
var db = monk('localhost:27017/nodetest1');
```

These lines tell our app we want to talk to MongoDB, we're going to use Monk to do it, and our database is located at `localhost:27017/nodetest1`. Note that 27017 is the default port your MongoDB instance should be running on. If for some reason you've changed it, obviously use that port instead. Now look further down in the file, where you have this:

C:\node\nodetest1\app.js

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

We need to do some work here. Those `app.use` statements (along with the others you'll find in `app.js`) are establishing middleware for Express. The short, simple explanation is: they're providing custom functions that the rest of your app can make use of. It's pretty straightforward, but due to chaining it needs to come *before* our route definitions, so that they can make use of it.

Above the two lines just mentioned, add the following:

C:\node\nodetest1\app.js

```
// Make our db accessible to our router
app.use(function(req,res,next){
    req.db = db;
    next();
});
```

NOTE: If you don't put this above the routing stuff mentioned above (`app.use('/', indexRouter);`), your app WILL NOT WORK.

We already defined `db` when we added Mongo and Monk to `app.js`. It's our Monk connection object. By adding this function to `app.use`, we're adding that object to every HTTP request (ie: `req`) our app makes. This is fine and will not cause performance issues. Unlike MySQL, MongoDB connections don't need to be manually opened and closed, so the db object will just be there, not using any resources, until we need it.

So, again, that code needs to go above our routing code. Your entire `app.js` should look like this, now:

C:\node\nodetest1\app.js

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

// New Code
var monk = require('monk');
var db = monk('localhost:27017/nodetest1');

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

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

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')));

// Make our db accessible to our router
app.use(function(req,res,next){
    req.db = db;
    next();
});

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

// 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');
});

module.exports = app;
```

Next thing we need to do is modify our route so that we can actually show data that's held in our database, using our `db` object.

Step 6 – Pull your data from Mongo and display it

Open up `C:\node\nodetest1\routes\index.js` in your editor. It's still got the index route, and the goofy /helloworld route. Let's add a third:

C:\node\nodetest1\routes\index.js

```
/* GET Userlist page. */
router.get('/userlist', function(req, res) {
    var db = req.db;
    var collection = db.get('usercollection');
    collection.find({},{},function(e,docs){
        res.render('userlist', {
            "userlist" : docs
        });
    });
});
```

OK ... that's getting fairly complicated. All it's really doing, though, is extracting the `db` object we passed to our http request, and then using that `db` connection to fill our `docs` variable with database documents, ie: user data. Then we do a page render just like the other two `GET`s in this route file.

Basically, we tell our app which collection we want to use (`usercollection`) and do a find, then return the results as the variable `docs`. Once we have those documents, we then do a render of `userlist` (which will need a corresponding EJS template), giving it the `userlist` variable to work with, and passing our database documents to that variable.

Next let's set up our EJS template. Navigate to `C:\node\nodetest1\views\` and open `index.ejs`. Once again, immediately save it as userlist.ejs. Then edit the HTML so it looks like this:

C:\node\nodetest1\view\userlist.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>User List</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1>User List</h1>
    <ul>
      <%
        var list = '';
        for (i = 0; i < userlist.length; i++) {
          list += '<li><a href="mailto:' + userlist[i].email + '">' + userlist[i].username + '</a>
        }
        return list;
      %>
    </ul>
  </body>
</html>
```

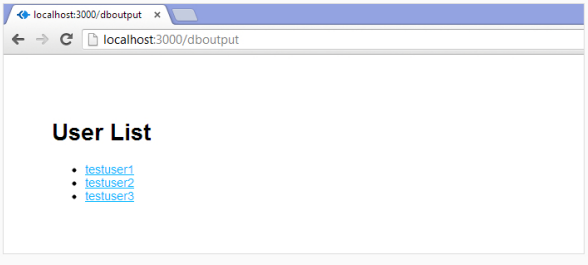
This is saying that we're going to pull in the set of documents we just called `userlist` over in the route file, and then for each entry, get the email and username values from the object and put them into our html. Oh, also, this seems like a good place to mention that we don't *have* to keep repeating the `<html>` tags and similar on every page. EJS supports including partial files. It's just outside of the scope of this tutorial.

We're all set. Save that file, and let's restart our node server. Remember how to do that? Go to your command prompt, head for `C:\node\nodetest1\` and `ctrl-c` or `cmd-c` to kill your server if it's still running from way back before. Then type:

Command C:\node\nodetest1\

```
C:\node\nodetest1>npm start
```

Now open your browser and head to http://localhost:3000/userlist and marvel at the results.



You're now pulling data from the DB and spitting it out onto a web page. Nice!

There one more thing I badly wanted to cover in this tutorial, but because it's already about as long as the Bible, I'm going to breeze through it here. You could very easily change your userlist view from an Express-driven web page complete with EJS template to a plain old JSON response. You could then access this with AJAX and

template to a plain old JSON response. You could then access this with AJAX and manipulate it on the client-side, with jQuery for example, instead of on the server side. In fact, I wanted to cover that so badly that I *wrote an entire second tutorial on it*. You can find the link at the end of this one!

Let's finish this up.

Part 4 – The holy grail: writing to the DB

Writing to the database is not particularly difficult. Essentially we need to set up a route that takes a `POST`, rather than a `GET`.

Step 1 – Create your data input

We're going quick and dirty here: two ugly, unstyled text inputs and a submit button. 1996-style, but before we get to that, we're going to do some javacripting. Let's start by quickly wiring up a route for our add user form. Open `/routes/index.js` and add the following code above the last module.exports line:

C:\node\nodetest1\routes\index.js

```
/* GET New User page. */
router.get('/newuser', function(req, res) {
    res.render('newuser', { title: 'Add New User' });
});
```

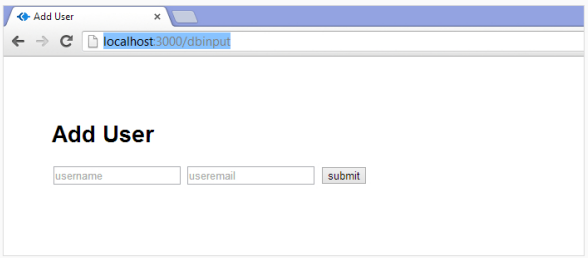
Now we just need a template. Open up /views/index.ejs, save it as newuser.ejs, and replace the whole file contents with this:

C:\node\nodetest1\views\newuser.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Add User</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <form id="formAddUser" name="adduser" method="post" action="/adduser">
      <input id="inputUserName" type="text" placeholder="username" name="username" />
      <input id="inputUserEmail" type="text" placeholder="email" name="useremail" />
      <button id="btnSubmit" type="submit">Submit</button>
    </form>
  </body>
</html>
```

Here we're creating a form with the ID `formAddUser` (I like to preface my IDs with the type of thing we're ID'ing. It's a personal quirk). Method is `post`, action is `adduser`. Pretty straightforward. Under that we've defined our two inputs and our button.

If you restart your node server and go to `http://localhost:3000/newuser` you'll see your form in all its glory.



Go ahead and submit. Enjoy the 404 error. We're about to fix that.

Step 2 – Create your DB functions

OK, this is pretty much the same process as with the user list. We don't need another app.use statement since we've already wrapped our database object into every request (see above). That means it'll be accessible to any new routes we write. That means that all we need to do is add a route for POSTing to /adduser.

Go back to `/routes/index.js` and let's create our insertion function. Once again, you'll want to put this above the final module.exports line (it doesn't REALLY matter, but it makes things cleaner to wrap up with the export). This is a big one, so I've commented the code pretty thoroughly. Here it is:

C:\node\nodetest1\routes\index.js

```
/* POST to Add User Service */
router.post('/adduser', function(req, res) {

    // Set our internal DB variable
    var db = req.db;

    // Get our form values. These rely on the "name" attributes
    var userName = req.body.username;
    var userEmail = req.body.useremail;

    // Set our collection
    var collection = db.get('usercollection');

    // Submit to the DB
    collection.insert({
        "username" : userName,
        "email" : userEmail
    }, function (err, doc) {
        if (err) {
            // If it failed, return error
            res.send("There was a problem adding the information to the database.");
        }
        else {
            // And forward to success page
            res.redirect("userlist");
        }
    });
});
```

Obviously in the real world you would want a *ton* more validating, error-checking, and the like. You'd want to check for duplicate usernames and emails, for example. And to vet that the email address at least looks like a legit entry. But this'll work for now. As you can see, on successfully adding to the DB, we forward the user back to the userlist page, where they should see their newly added user.

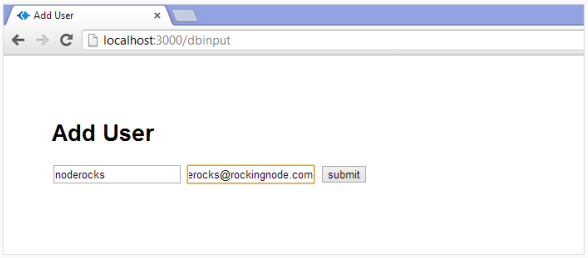
Are there smoother ways to do this? Absolutely. We're staying as bare-bones as possible here. Now, let's go add some data!

Step 3 – Connect and add data to your DB

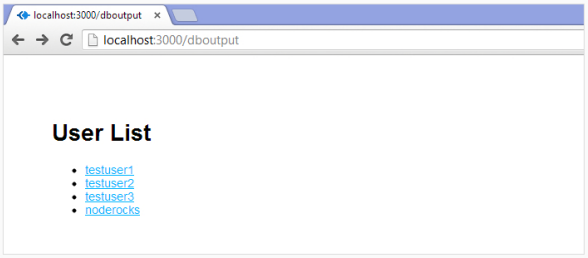
Head to your command prompt, kill your node server if it's still running, and restart it:

```
Command C:\node\nodetest1\

C:\node\nodetest1>npm start
```

Assuming your server is running, which it should be, return to your web browser and point it at http://localhost:3000/newuser again. There's our exciting form, just like before. Except now let's fill in some values before we hit submit. I went with username "noderocks" and email "noderocks@rockingnode.com" ... you can go with whatever you'd like.



Click submit, and check it out ... we're back at /userlist and there's our new entry!



We are officially reading and writing from a MongoDB database using Node.js, Express, and EJS. You are now what the kids call a "full stack" developer (probably not a GOOD one, just yet, but I didn't promise that).

Congratulations. Seriously. If you followed this all the way through, and if you really paid attention to what you were doing and didn't just paste code, you should have a really solid grasp on routes and views, reading from the DB, and posting to the DB. That is all you need to get started developing whatever app you want to build. I don't know about you, but I think that's really cool.

Part 5 – Next steps

The "sequel" to this tutorial is now available! Find it at: Creating a Simple RESTful Web App with Node.js, Express, and MongoDB.

From here, there's a million different directions you could go. I recommend completing the tutorial mentioned above, or you could dive into our Five Minute React series, which will step you through building and deploying a barebones web application using a custom Express.js API and a React front-end.

I hope this tutorial's been helpful. I wrote it because I could've used it when I got started, and I couldn't seem to find something that was quite at this level, or that broke things down in such long, long, *loooong* detail. If you made it this far, thanks for sticking with it!