

CWBUECHELER (/)

HOME (/) WEB (/WEB/) WRITING (/WRITING/)
COCKTAILS (/COCKTAILS/) BEER (/BEER/)
VIDEO GAMES (/VIDEOGAMES/) MUSIC (/MUSIC/)
SPORTS (/SPORTS/) CONTACT (/CONTACT/)



CREATING A SIMPLE RESTFUL WEB APP WITH NODE.JS, EXPRESS, AND MONGODB

LEARN THE BASICS OF REST AND USE
THEM TO BUILD AN EASY, FAST,
SINGLE-PAGE WEB APP.

By Christopher Buecheler
(<http://cwbuecheler.com>)

NEW: Updated for ExpressJS v4!

You can find/fork the sample project on
GitHub
(<https://github.com/cwbuecheler/node-tutorial-2-restful-app>)

In my first tutorial
(<http://cwbuecheler.com/web/tutorials/2013/node-express-mongo/>), we looked at how to go from
nothing installed, to a fully-functioning Node.js

web app, using the Express framework, that reads from and writes to a MongoDB database. That's a great start, and if you're unfamiliar with those technologies, now would be an excellent time to go through the tutorial, because we're about to delve deeper. You're going to need to know how to get a webserver running with Express, and how to use `app.get` and `app.post` to communicate with both the server and the database. It's all covered in that original tutorial, and if you're a developer who's familiar with JavaScript, it's not hard. Go check it out (<http://cwbuecheler.com/web/tutorials/2013/node-express-mongo/>)!

Back? ... or rolling your eyes and going, "dude, I already know that stuff"? Either way, great! Let's add some new tools to our toolbox, and create a simple little app that works without a single page refresh. Here are the goals:

- Learn what REST means in plain English
- Store and retrieve JSON data in a MongoDB collection using HTTP POST and HTTP GET
- Remove data from the collection using HTTP DELETE
- Use AJAX for all data operations
- Update the DOM with jQuery

Basically, in the first tutorial, we built a simple front-end app atop the Router/View back-end. In this tutorial, we're going to eliminate the need for page refreshing or visiting separate URIs entirely. It's all going to work out fine. But before we start building, let's get some REST ...

PART 1 - SERIOUSLY, WHAT THE HELL IS REST?

Wikipedia, that infallible source, defines Representational State Transfer (REST) as: *an architectural style that abstracts the architectural elements within a distributed hypermedia system*. Everyone got that? We clear and ready to move on?

... not so much?

Yeah, me either. I have no idea what a distributed hypermedia system is, and I'm OK admitting that. Let's try to put the concepts of REST in plain English. To do that, we're going to borrow four basic design principles from IBM's developerWorks website (<http://www.ibm.com/developerworks/webservices/restful/>), and explain what they mean.

- Use HTTP methods explicitly.
- Be stateless.
- Expose directory structure-like URIs.
- Transfer XML, JavaScript Object Notation (JSON), or both.

Use HTTP Methods Explicitly

This one's pretty straightforward. To retrieve data, you use GET. To create data, you use POST. To update or change data, you use PUT (not used in this tutorial). To delete data you use DELETE. So for example, this once-common approach is not a good one:

`http://www.domain.com/myservice/newuser.php?newuser=bob`

That's an HTTP GET pretending to be a POST. You're GETting the web page and giving it data to store in a DB at the same time. Instead, create a `NewUser` service and POST to it.

Be Stateless

This is a complicated concept but it boils down to "don't store state information on the server". If you must save state, save it on the client side via cookies or other methods. A front-end framework like Angular (outside the scope of this tutorial, but stay tuned!) is particularly helpful here, as it creates an entire client-side MVC setup where you can save and manipulate the state of elements without hammering your server.

IBM gives a pagination example which is pretty good. A stateful design would hit a `deliverPage` service that's been keeping track of the page you're on, and delivers the next one. a Stateless design would populate `prevPage`, `currPage`, and

nextPage data in the markup (hidden input fields, JavaScript variables, data- attributes, and so on), and then HTTP GET a newPage service using the nextPage parameter from the markup to request a specific page.

I've put together a quick JSFiddle (<http://jsfiddle.net/cwbuecheler/7fars/>) illustrating what I'm talking about. Take a look and note that we're never storing any page data on the "server" side. We merely take the current page value from the DOM, and then when we get our new page, we update the DOM. That's simple, stateless programming.

Expose directory structure-like URIs.

This one's easy. Instead of:

`http://app.com/getfile.php?
type=video&game=skyrim&pid=68`

You want:

`http://app.com/files/video/skyrim/68`

Transfer XML, JavaScript Object Notation (JSON), or both.

This one's easy too! Just make sure that your back-end is sending XML or JSON (I prefer JSON, especially in all-JavaScript setups like the one discussed in this tutorial). You can easily manipulate this data in your presentation layer without having to hit your servers, unless you need new data.

OK ... so do we get the basics of REST? It's pretty straightforward, really. You've probably already worked within systems that use it.

PART 2 - SETUP

Now that we have an idea of what REST is all about, let's put it to work for us by building a stupidly simple single-page web app that's completely valueless. Kind of like half the startups in the Valley, amirite?! Is this thing on?

Anyway, no, we're not building a to-do list, though that's become the "Hello World" of web apps. We're going to build a simple collection of usernames and emails, much like we did in our previous tutorial. However, a few things need to be set up, and also we're going to tear down one thing: Monk. We're switching over to Mongoskin for our MongoDB management. Why? Well, for one thing, nothing significant's been committed to the Monk Github page in about a year, and that makes me nervous. For another, Mongoskin is lightweight and doesn't require you to define schemas the way Mongoose does, but it's a little more full-featured than Monk. The syntax is almost identical.

So, let's get started. Make sure you've got the latest stable version of Node installed on your machine, then fire up a console window and navigate to wherever you're storing these web projects. For the purposes of this tutorial, that's C:\node\. If you're placing your work elsewhere (for example /home or /Users), adjust accordingly.

The first thing we'll want to do is update Express and the Express scaffolding generator globally, like this:

```
COMMAND C:\NODE\
```

```
npm update -g express
```

Once that's done, type the following:

```
COMMAND C:\NODE\
```

```
npm update -g generator-express
```

Finally, follow it up with:

```
COMMAND C:\NODE\
```

```
express nodetest2
```

As you'll remember from this last tutorial, this is going to auto-generate a website skeleton in a new directory called nodetest2. Watch as it does its thing, and when it's done, open the newly created package.json file (in the newly created nodetest2 folder) in your text editor of choice, and change it so it looks like this:

C:\NODE\NODETEST2\PACKAGE.JSON

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "~4.0.0",
    "static-favicon": "~1.0.0",
    "morgan": "~1.0.0",
    "cookie-parser": "~1.0.1",
    "body-parser": "~1.0.0",
    "debug": "~0.7.4",
    "jade": "*",
    "mongodb": "*",
    "mongoskin": "*"
  }
}
```

The asterisks mean “just gimme the latest version” and are generally all you need. Note that we're adding the MongoDB and Mongoskin packages so that we can access and control our database. Switch back to your command prompt, **cd to your nodetest2 directory**, and type:

COMMAND C:\NODE\NODETEST2\

```
npm install
```

Watch as all your dependencies install – it'll take a while. Very exciting! After that, there's one small thing to do. In the same command prompt, just type:

```
COMMAND C:\NODE\NODETEST2\
```

```
mkdir data
```

This is where we'll store our database files when we get to that point. If you'd like to store them elsewhere, that's absolutely fine ... just know that you need to have the directory ready before running MongoDB. That's it; we're going to prepopulate the database later in this tutorial, so we're done with setup for now. It's time to get started making web pages.

One final note: this article uses four-space indentation for everything because that's fairly popular in the JavaScript community. I actually mostly work in tabs. I know this is a big debate, but honestly given how fast and easy it is to convert spaces to tabs and back again with any decent editor, I don't really see the issue. Unless your team has a standard everyone's agreed to, you should work with what you want.

PART 3 - STARTING IN ON HTML

If we're going to have a single-page web app, the first thing we need is a single page, right? Let's open up our views folder, and start with layout.jade. This is a template file that we're only going to make a few basic changes to. Here's how it starts:

```
C:\NODE\NODETEST2\VIEWS\LAYOUT.JADE
```

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

We want to be able to do two things. One: include jQuery, and two: include our master javascript file. So let's edit the file and make it look like this:

C:\NODE\NODETEST2\VIEWS\LAYOUT.JADE

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
    script(src='http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js')
    script(src='/javascripts/global.js')
```

Astute readers may note that global.js does not actually exist yet. This is true. We're going to make it in a bit. For now, it'll just quietly 404 in the background when we load our index page. If this really bothers you, feel free to create an empty file now in /public/javascripts/.

Onward to index.jade – this is the only HTML file we'll need for the rest of our webapp. We're going to put quite a lot of stuff onto this page. In the real world, we'd also need quite a lot of CSS, but for the sake of saving time, just download this file (<http://cwbuecheler.com/web/tutorials/2014/restful-web-app-node-express-mongodb/style.css>) and copy it over /public/stylesheets/style.css. It'll give you a basic layout to work with, which you can modify to be as pretty as your heart desires.

Open index.jade. You'll see a VERY basic skeleton here:

C:\NODE\NODETEST2\VIEWS\INDEX.JADE

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

That's not very exciting. Let's get started modifying this guy. That h1= title line, and the paragraph after it, are both pulling a title variable that's set in /routes/index.js, and just says "Express". That's kind of repetetive, so let's change the paragraph

to just say "Welcome to our test". Then let's add a wrapper, and add a table skeleton for displaying a list of users:

C:\NODE\NODETEST2\VIEWS\INDEX.JADE

```
extends layout

block content
  h1= title
  p Welcome to our test

  // Wrapper
  #wrapper

    // USER LIST
    h2 User List
    #userList
      table
        thead
          th UserName
          th Email
          th Delete?
        tbody
      // /USER LIST
    // /WRAPPER
```

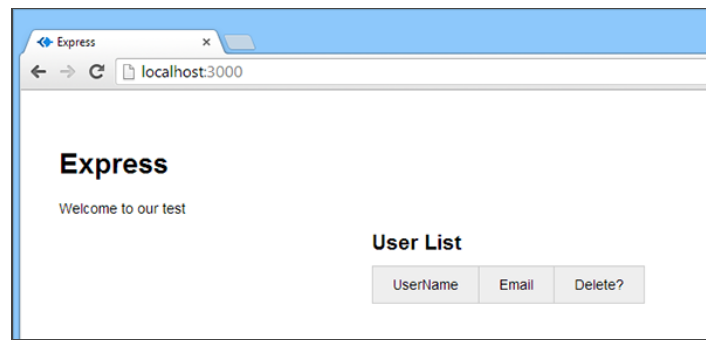
You'll note that there's no data in the table. That's because we're going to populate it, via AJAX, with stuff we pull out of our MongoDB database. So, let's get that set up, and then we'll do a little javascripting to pull things together.

If you want to see what the page looks like right now, go ahead and type the following into your terminal:

COMMAND C:\NODE\NODETEST2\

```
npm start
```

Then navigate to <http://localhost:3000> (<http://localhost:3000>) ... you'll see a very basic, very boring page with an empty table in it. If you've downloaded the CSS file mentioned above and overwritten the default stylesheet, it will look like this:



Exciting, right? Not really. Let's make it do something.

PART 4 - THE DATABASE

We already covered getting MongoDB up and running in the previous tutorial, so I'm going to skip right to the good stuff here. In a **new** terminal or command prompt window, navigate to wherever you keep MongoDB (for example: C:\mongo) and type

```
COMMAND C:\MONGO\
```

```
mongod - -dbpath c:\node\nodetest2\data
```

Obviously if you've decided to store your data elsewhere, you should use that path instead. You should see the MongoDB daemon fire up and report that it's waiting for connections. So let's connect: open up a **new** terminal/command window, navigate to your Mongo directory, and type:

```
COMMAND C:\MONGO\
```

```
mongo
```

Now we've got a command interface for our database. This, too, was discussed in the previous tutorial, so again I'm going to jump right into "just doing" instead of explaining. Let's switch to a new database (for our new project) by typing the following:

```
MONGO CONSOLE
```

```
use nodetest2
```

Now we want to create and populate the “userlist” collection within our nodetest2 database. We're going to do this in one fell swoop by just inserting into the empty collection, like this:

MONGO CONSOLE

```
db.userlist.insert({'username' : 'test1', 'email' : 'test1@example.com'})
```

Note: I like to type this stuff out in a text editor first, using tabs and all that, so I can see what I'm doing, before collapsing it down to a single line and pasting it into the MongoDB console. You should do whatever works best for you.

That's all the data we need to prepopulate, this time around. We don't even *really* need to do that much, since we're going to be creating an add user routine anyway, but it makes it easier to see that yes, our DB connection is working, as we first start to wire up our app.

You can exit out of the MongoDB console right now and kill that terminal, if you want, but make sure to leave the MongoDB daemon running. If you close it down, our website won't be able to connect to a database, and that would be a Bad Thing.

PART 5 - LISTING USERS

It's time to start making changes to app.js, the heart and soul of our application. By default Express has set it up pretty well, but we'll need to add some hooks for Mongoskin. Here's what we start with:

```
C:\NODE\NODETEST2\APP.JS
```

```

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

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

var app = express();

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

app.use(favicon());
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser());
app.use(express.static(path.join(__dirname,

app.use('/', routes);
app.use('/users', users);

/// catch 404 and forwarding to error handle
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

/// error handlers

// development error handler
// will print stacktrace
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});

module.exports = app;

```

We're going to add a few things to the section at the very top, so that it looks like this:

UPDATE: the syntax for Mongoskin has changed somewhat since this tutorial was posted. The code below reflects the **NEW** syntax for Mongoskin versions 1.3.20-alpha and higher.

C:\NODE\NODETEST2\APP.JS

```
var express = require('express');
var path = require('path');
var favicon = require('static-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
// Database
var mongo = require('mongoskin');
var db = mongo.db("mongodb://localhost:27017/");
```

Here we're calling the Mongoskin module and then giving it some basic configuration parameters (including telling it where the DB lives, and which database to use – nodetest2).

We also need to make our database accessible to our various http requests, as we did in the first tutorial. To do that, first find this section:

C:\NODE\NODETEST2\APP.JS

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

And just **above it**, add this code:

C:\NODE\NODETEST2\APP.JS

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

Now it's time to move on to routing. Note that Express auto-creates a /users route file. We're going to make use of that, but we won't be creating any views for it. Why? Well, because since this is a single-page app, we're using the Index

route and view for display purposes. We're going to use the user route to set up our data I/O ... the services we want to create to show, add, and delete users from our database. We'll access these with JavaScript, rather than navigating to them in a browser, and display the collected data on the index page.

So to that end, let's do a little cleanup. Open up `/nodetest2/routes/users.js` in your text editor. It'll look like this:

C:\NODE\NODETEST2\ROUTES\USERS.JS

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

/* GET users listing. */
router.get('/', function(req, res) {
  res.send('respond with a resource');
});

module.exports = router;
```

Go ahead and delete the comment and the three lines below it. We're going to replace it with our own code, for `/users/userlist`. So our file's going to look like this:

C:\NODE\NODETEST2\ROUTES\USERS.JS

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

/*
 * GET userlist.
 */
router.get('/userlist', function(req, res) {
  var db = req.db;
  db.collection('userlist').find().toArray(function(err, items) {
    res.json(items);
  });
});

module.exports = router;
```

The purpose of this code is: if you do an HTTP GET to `/users/userlist`, our server will return JSON that lists all of the users in the database. Obviously for a large-scale project you'd want to put in limits as

to how much data gets spewed out at one time, for example by adding paging to your front-end, but for our purposes this is fine.

Save your users.js file, kill your node instance if it's still running, and restart it with

```
COMMAND C:\NODE\NODETEST2\
```

```
npm start
```

In your terminal. Then refresh your browser. You'll see ... nothing. Well, not nothing, but rather the exact same thing as you saw in your screenshot above. That's because we haven't wired anything up yet. If you want, you can navigate to **<http://localhost:3000/users/userlist>** (**<http://localhost:3000/users/userlist>**) where you will find the JSON output that we'll be manipulating next. It's just a single user, the one that we manually entered in the MongoDB console earlier.

Let's get Bob Smith into our HTML, shall we? We're going to create our global.js file now, so create a new text document and save it as `/nodetest2/public/javascripts/global.js`

A few notes on my coding style: I like braces and use them always, even for single-line if statements and the like. I like variable names that mean something, rather than trying to be short. I favor single quotes above double quotes. I prefer comments above lines, as opposed to on the right-hand side. I use a LOT of comments, because JS can always be minified so there's really no reason to worry about comments contributing to filesize. I use a decent amount of whitespace for the same reason. Oh, and I like my opening braces on the same line, not a new line.

You probably hate one or more things about my style. That's fine. Adapt as you see fit. What matters is that your code is readable (if you're

working with others) and that it runs (no matter what).

Let's start by defining a function to populate our HTML table with data. I like to label the different sections of my JavaScript with big, hideous, highly-noticeable labels, so here's what we're going to do:

C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J

```
// Userlist data array for filling in info
var userListData = [];

// DOM Ready =====
$(document).ready(function() {

    // Populate the user table on initial page load
    populateTable();

});

// Functions =====

// Fill table with data
function populateTable() {

    // Empty content string
    var tableContent = '';

    // jQuery AJAX call for JSON
    $.getJSON( '/users/userlist', function(data) {

        // For each item in our JSON, add a row to the table
        $.each(data, function(i, user) {
            tableContent += '<tr>';
            tableContent += '<td><a href="#">' + user.name + '</a>';
            tableContent += '<td>' + user.email + '</td>';
            tableContent += '<td><a href="#">' + user.username + '</a>';
            tableContent += '</tr>';
        });

        // Inject the whole content string into the table body
        $('#userList table tbody').html(tableContent);

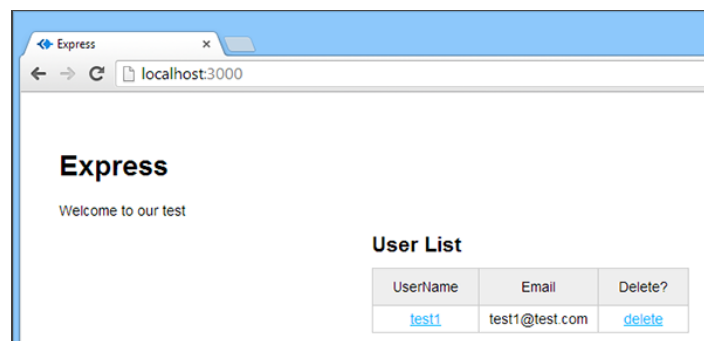
    });
};
```

As you can see, we're using a global variable and defining our functions at the top level. This is actually not a great idea and I'm only doing it for speed and simplicity. In a real app, you want to define a single master global, which you can then populate with properties, methods, etc ... as needed, thus helping to avoid conflicts or generally pollute the namespace. If you want to

learn more about this strategy, I strongly recommend picking up a copy of JavaScript: The Good Parts (<http://www.amazon.com/JavaScript-Good-Parts-Douglas-Crockford/dp/0596517742>) by Douglas Crockford and reading through it. Actually, I strongly recommend that anyway. It's an important book.

For now, however, we'll stick with our specific global and move on to the DOM ready detect, which will fire off our table-filling method *populateTable()* when the page is ready for scripts to run. After that, we must of course define our table-filling method. That's where things get interesting, but not too terribly complex. We make a simple AJAX call via jQuery, iterate over the return JSON to create a big ol' content string with all our new HTML in it, and then inject that HTML into our existing table.

With that JavaScript added, we can now go to our browser again and refresh or navigate to <http://localhost:3000/> (<http://localhost:3000/>) ... if all is working properly, and it should be, we'll see that our table is now populated with data. Awesome!



That's a solid start, but it's not displaying everything we have in our database. Let's set up an info box that will display the full set of user information when we click on the username.

PART 6 – POPULATING USER INFO

Still working in *global.js*, we need to add one quick line to our *populateTable()* function. This line will stick *all* of our user data into the array we established earlier. Again: I don't recommend

taking this route if you're dealing with thousands of users. It's not a performance-friendly approach. But for quick-n-dirty, it'll work fine.

Find this line:

```
C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J
```

```
// jQuery AJAX call for JSON  
$.getJSON( '/users/userlist', function(
```

And directly below it you'll see:

```
C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J
```

```
// For each item in our JSON, add a tab  
$.each(data, function(){
```

In between those lines, add the following:

```
C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J
```

```
// Stick our user data array into a user  
userListData = data;
```

What this is doing is sticking all of our returned user data, from the database, into our global variable, so that we can access it without repeatedly whaling on the database each time we click a name in our table. I want to be clear here: for large-scale operations, this is **not a great idea**. You would not want to do this if you were loading tens of thousands of users at once (but you probably wouldn't want to load tens of thousands of users at once, either). You'd want to implement paging, and only load the data you really needed at any given time.

But for the purposes of this tutorial, it's fine. So head down to the end of the file, where we're keeping our functions, and add this code on a new line at the bottom:

C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J

```
// Show User Info
function showUserInfo(event) {

    // Prevent Link from Firing
    event.preventDefault();

    // Retrieve username from link rel attr
    var thisUserName = $(this).attr('rel');

    // Get Index of object based on id value
    var arrayPosition = userListData.map(fur
```

OK, that's kinda some crazy stuff. Here's what we're doing: first we're using `.map` to apply a function to each object in our `userListData` array. This will spit out a **brand new array** containing only whatever the function returns. That function (the anonymous callback function using the `userObj` parameter) strictly returns the username. So, basically, if our original data array contained two complete user objects, then the array returned by our use of `.map` here would only contain usernames, and look like this: `['Bob', 'Sue']`.

So once we have THAT array, provided by `.map`, we're chaining `indexOf`, in combination with the username of our choice, to get the array index of that username. So Bob would be zero, and Sue would be one. We can then use that number, stored as `arrayPosition`, to go back to our original user data array and start pulling data, in the following code.

If you want to see a version of this code that DOESN'T use chaining and is a bit more verbose, check it out at this jsfiddle (<http://jsfiddle.net/cwbuecheler/2qEZ2/>). In the meantime, still in that `showUserInfo` function, add the rest:

C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J

```

        // Get our User Object
        var thisUserObject = userListData[arrayI

        //Populate Info Box
        $('#userInfoName').text(thisUserObject.name);
        $('#userInfoAge').text(thisUserObject.age);
        $('#userInfoGender').text(thisUserObject.gender);
        $('#userInfoLocation').text(thisUserObject.location);

    };

```

The function in its entirety will look like this:

```

// Show User Info
function showUserInfo(event) {

    // Prevent Link from Firing
    event.preventDefault();

    // Retrieve username from link rel attribute
    var thisUserName = $(this).attr('rel');

    // Get Index of object based on id value
    var arrayPosition = userListData.map(function(object, index) {
        if (object.id === thisUserName) {
            return index;
        }
    });

    // Get our User Object
    var thisUserObject = userListData[arrayPosition];

    //Populate Info Box
    $('#userInfoName').text(thisUserObject.name);
    $('#userInfoAge').text(thisUserObject.age);
    $('#userInfoGender').text(thisUserObject.gender);
    $('#userInfoLocation').text(thisUserObject.location);

};

```

Now we need to trigger that function on a click, so back up in our DOM ready section, below the initial call to populate the table, add this code:

```

// Username link click
$('#userList table tbody').on('click',

```

This is all pretty straightforward. We've found our user object and we're populating spans with data. The thing is ... those spans don't exist yet! Let's

save **global.js** and then open up `/views/index.jade`. We want to add the following code right after the `#wrapper` line (and above our user list table):

```
C:\NODE\NODETEST2\VIEWS\INDEX.JADE

// USER INFO
#userInfo
  h2 User Info
  p
    strong Name:
    | <span id='userInfoName'><
    br
    strong Age:
    | <span id='userInfoAge'><
    br
    strong Gender:
    | <span id='userInfoGender'
    br
    strong Location:
    | <span id='userInfoLocati
// /USER INFO
```

Remember that the indents here are really important – if your indents are borked, Jade flat-out won't work. For reference, here's what your entire `index.jade` file should look like right now:

```
C:\NODE\NODETEST2\VIEWS\INDEX.JADE
```

```

extends layout

block content
  h1= title
  p Welcome to our test

  // Wrapper
  #wrapper

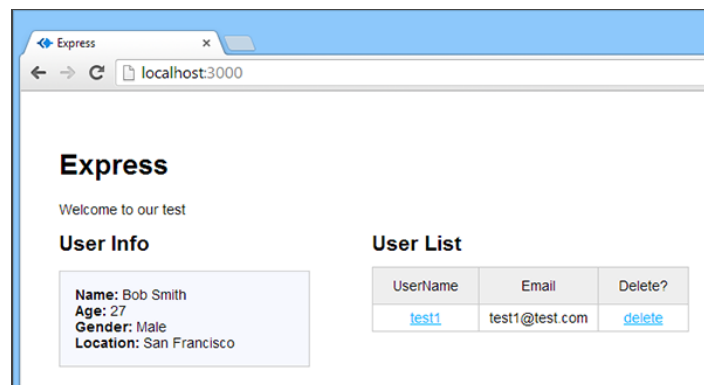
    // USER INFO
    #userInfo
      h2 User Info
      p
        strong Name:
        | <span id='userInfoName'><
        br
        strong Age:
        | <span id='userInfoAge'><
        br
        strong Gender:
        | <span id='userInfoGender'
        br
        strong Location:
        | <span id='userInfoLocati
    // /USER INFO

    // USER LIST
    h2 User List
    #userList
      table
        thead
          th UserName
          th Email
          th Delete?
        tbody
    // /USER LIST

  // /WRAPPER

```

Got it? Cool. Save that file and let's hit our browser and take a look. Refresh <http://localhost:3000> (<http://localhost:3000>) and now you'll see an unpopulated user info box floating over to the left. Click on the "test1" username in your table, and it should populate that box with Bob's info.



Pretty awesome. So now let's get some more users in there by doing some AJAX posting.

PART 7 - ADDING USERS

The first thing we'll need is a set of form fields with which to add a new user. So let's stick around in `/views/index.jade` for a bit and add that code. Right below our user list, but above the closing wrapper comment, add the following:

C:\NODE\NODETEST2\VIEWS\INDEX.JADE

```
// ADD USER
h2 Add User
#addUser
  fieldset
    input#inputUserName(type='text')
    input#inputEmail(type='text')
    br
    input#inputUserFullname(type='text')
    input#inputUserAge(type='text')
    br
    input#inputUserLocation(type='text')
    input#inputUserGender(type='text')
    br
    button#btnAddUser Add User
// /ADD USER
```

This is all pretty straightforward (yes, I'm using a couple of break tags, which are best avoided outside of text, but it's for the sake of expediency here). We've got text inputs for each piece of information we'll need to create a new user, and we've got a button to click on. Right now that button won't do anything, but that's what we're about to fix.

This is the last time we'll need to touch our view file. Make sure it looks like this in your text editor:

C:\NODE\NODETEST2\VIEWS\INDEX.JADE

```

extends layout

block content
    h1= title
    p Welcome to our test

    // Wrapper
    #wrapper

        // USER INFO
        #userInfo
            h2 User Info
            p
                strong Name:
                | <span id='userInfoName'><
                br
                strong Age:
                | <span id='userInfoAge'><
                br
                strong Gender:
                | <span id='userInfoGender'
                br
                strong Location:
                | <span id='userInfoLocati

        // /USER INFO

        // USER LIST
        h2 User List
        #userList
            table
                thead
                    th UserName
                    th Email
                    th Delete?

                tbody

        // /USER LIST

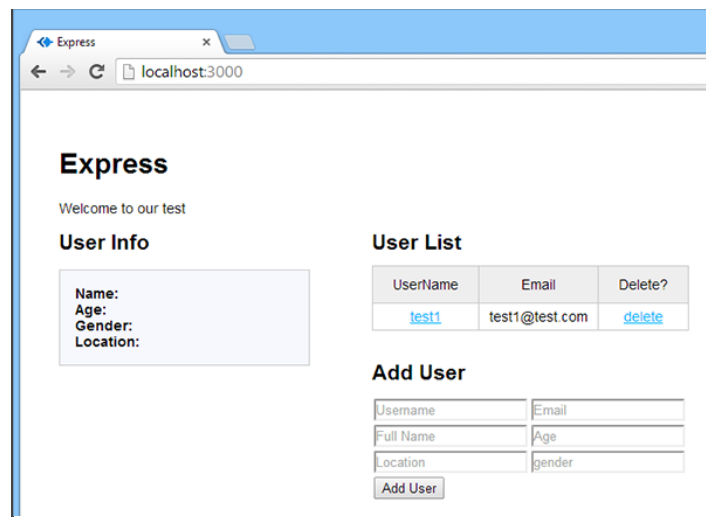
        // ADD USER
        h2 Add User
        #addUser
            fieldset
                input#inputUserName(type='te
                input#inputUserEmail(type='t
                br
                input#inputUserFullname(type
                input#inputUserAge(type='te
                br
                input#inputUserLocation(type
                input#inputUserGender(type=
                br
                button#btnAddUser Add User

        // /ADD USER

    // /WRAPPER

```

And like this in your browser:



One last reminder: Jade is very touchy about tabs/spaces. If you're getting errors, check all of your indents and try restarting app.js. If everything looks right, you're good to go. Close index.jade, and open /routes/users.js – it's time to do some POSTing.

Adding a user is really not too difficult. It's very similar to retrieving user info, except you're doing an insert() to the database instead of a find(). Below your user listing code (but above module.exports), you'll want to add the following:

C:\NODE\NODETEST2\ROUTES\USERS.JS

```
/*
 * POST to adduser.
 */
router.post('/adduser', function(req, res) {
  var db = req.db;
  db.collection('userlist').insert(req.body, function(err, res) {
    res.send(
      (err === null) ? { msg: '' } : { msg: err.message }
    );
  });
});
```

This basically just says “we’re going to post some data (req.body), and you’re going to insert it into our ‘userlist’ collection in the database. If that goes well, return an empty string. If it goes poorly, return the error message that the database gives us.”

For the purposes of this tutorial, that's really all there is to it. You can save /routes/users.js. That's all we need to do here. Now all that's left is making that new form/button do something. Head over to global.js and let's get started.

First let's write our event catcher, up in the DOM Ready section, just below the user list name click. It's just one line of code:

C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J

```
// Add User button click
$('#btnAddUser').on('click', addUser);
```

You'll note that we're calling an addUser function. Obviously, we'll need to build that function. This is going to be our largest single bit of code in this entire tutorial, because it's got quite a bit going on. It has to do some rudimentary form validation and then, if the form's all filled out, compile the data and POST it via AJAX to our adduser service.

This is a big function. I've commented it thoroughly to show what's going on. Add this at the bottom of global.js:

C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J

```

// Add User
function addUser(event) {
    event.preventDefault();

    // Super basic validation - increase error count
    var errorCount = 0;
    $('#addUser input').each(function(index, element) {
        if($(this).val() === '') { errorCount++; }
    });

    // Check and make sure errorCount is still 0
    if(errorCount === 0) {

        // If it is, compile all user info into an object
        var newUser = {
            'username': $('#addUser fieldset input:first').val(),
            'email': $('#addUser fieldset input:nth(2)').val(),
            'fullname': $('#addUser fieldset input:nth(3)').val(),
            'age': $('#addUser fieldset input:nth(4)').val(),
            'location': $('#addUser fieldset input:nth(5)').val(),
            'gender': $('#addUser fieldset input:last').val()
        };

        // Use AJAX to post the object to our server
        $.ajax({
            type: 'POST',
            data: newUser,
            url: '/users/adduser',
            dataType: 'JSON'
        }).done(function( response ) {

            // Check for successful (blank) response
            if (response.msg === '') {

                // Clear the form inputs
                $('#addUser fieldset input').val('');

                // Update the table
                populateTable();

            }
            else {

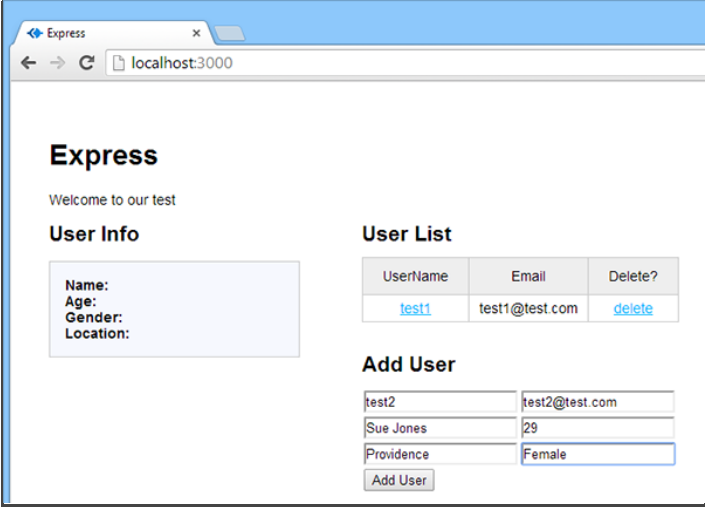
                // If something goes wrong,
                alert('Error: ' + response.r);

            }
        });
    }
    else {
        // If errorCount is more than 0, error
        alert('Please fill in all fields');
        return false;
    }
};

```

I recommend typing all that out instead of cutting and pasting it – it'll help you see what's really going on in a way that just scanning it with your eyes won't. But either way, get that into your

javascript and save the file. Now remember to **restart your node app** and then refresh `http://localhost:3000` (`http://localhost:3000`) ... you'll see the same thing you saw before, except now that form should work! Let's fill in some info.



Express

Welcome to our test

User Info

Name:
Age:
Gender:
Location:

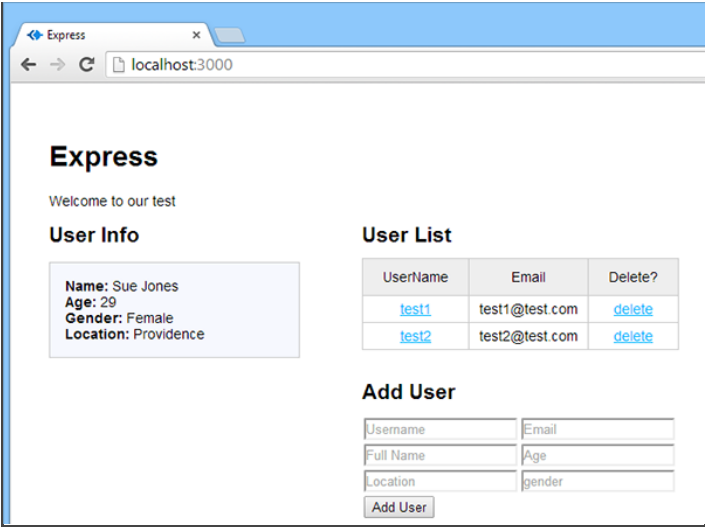
User List

UserName	Email	Delete?
test1	test1@test.com	delete

Add User

test2 test2@test.com
Sue Jones 29
Providence Female

Click add, and the form should submit, the data should be sent off to the database, and the table should refresh to show this fact. Click on your new username in that table, and you'll get your new user info in the info box. You'll note that all of this happened without any kind of a page refresh, which is what we're looking for here.



Express

Welcome to our test

User Info

Name: Sue Jones
Age: 29
Gender: Female
Location: Providence

User List

UserName	Email	Delete?
test1	test1@test.com	delete
test2	test2@test.com	delete

Add User

Username Email
Full Name Age
Location gender

Go ahead and create a few more users. We're going to be getting rid of a couple of 'em in the next section.

PART 8 - DELETING USERS

This is easier than adding users, but it's largely the same process: update our route file and update global.js ... we don't even have to touch index.jade because we already put our delete links in there. Good times.

Let's start with the route file. **Update:** the syntax has changed since I first posted this tutorial. The **NEW**, correct syntax is shown here. Open /routes/users.js and add the following to the bottom, just above module.exports:

C:\NODE\NODETEST2\ROUTES\USERS.JS

```
/*
 * DELETE to deleteuser.
 */
router.delete('/deleteuser/:id', function(req, res) {
  var db = req.db;
  var userToDelete = req.params.id;
  db.collection('userlist').removeById(userToDelete, function(err, result) {
    res.send((result === 1) ? { msg: 'deleted' } : { msg: 'error' });
  });
});
```

This is pretty straightforward – we pass in an ID parameter (so the URI we'll be referencing will be /deleteuser/12345 for example, and we reference it in the code with req.params.id), and MongoDB matches it up with the unique _id field that it generates for every entry in a collection, and nukes that entry from orbit. Just like our add user routine, if all goes well it returns a blank string, and if things don't work out it sends back the error message from MongoDB.

Save /routes/users.js and then we'll wire stuff up in the JavaScript. Let's move back to global.js to finish things up. We've already populated each delete link with a rel attribute that contains the id. It's in this line from global.js:

C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.JS

```
tableContent += '<td><a href="#" class="link delete" rel="' + user._id + '">Delete</a>';
```

Now we're going to add a quick delete routine in our DOM Ready section:

```
C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J
```

```
// Delete User link click  
$('#userList table tbody').on('click',
```

Note the syntax we're using: when working with jQuery's 'on' method, in order to capture dynamically inserted links, you need to reference a static element on the page first. That's why our selector is the table's tbody element – which remains constant regardless of adding or removing users – and then we're specifying the specific links we're trying to catch in the .on parameters.

Let's build that deleteUser function down at the bottom of our file:

```
C:\NODE\NODETEST2\PUBLIC\JAVASCRIPTS\GLOBAL.J
```

```

// Delete User
function deleteUser(event) {

    event.preventDefault();

    // Pop up a confirmation dialog
    var confirmation = confirm('Are you sure');

    // Check and make sure the user confirmed
    if (confirmation === true) {

        // If they did, do our delete
        $.ajax({
            type: 'DELETE',
            url: '/users/deleteuser/' + $(this).attr('id'),
        }).done(function( response ) {

            // Check for a successful (blank) response
            if (response.msg === '') {

                // Success
            }
            else {
                alert('Error: ' + response.msg);
            }

            // Update the table
            populateTable();

        });

    }
    else {

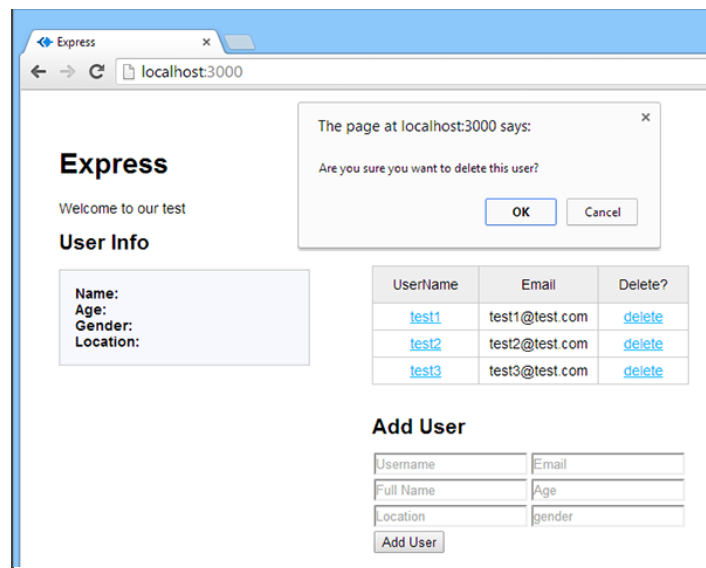
        // If they said no to the confirm, do nothing
        return false;

    }

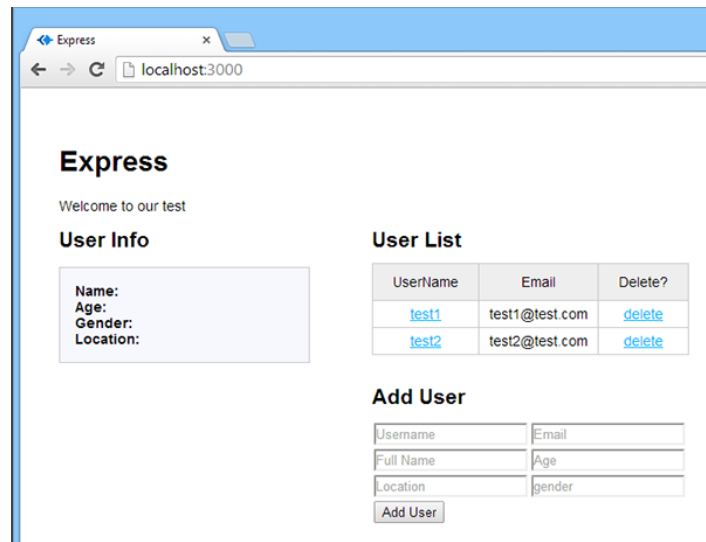
};

```

Remember to **restart your node server** and then hit `http://localhost:3000` to test it out. This one's more simple than the `addUser` function. It does a quick JavaScript confirm:



If the user's sure they want to delete, it just hits our delete service and passes it the ID, then it updates the table to show that the user is now gone.



That's it. We've got delete working. You can now add, view, and delete as many users as you'd like. Since this tutorial's already the length of a dictionary, we're going to skip updating, but hopefully you have a solid idea of how to do that. First you'd GET the info and populate a form with it, then you'd PUT on submission of the form (see the beginning of this tutorial for info on POST vs. PUT), and update the table.

Actually, that's not a bad exercise: you should try to add updating user info to this page and see if you can get it working. You'll need to edit

/views/index.jade, /routes/users.js, /app.js, and /public/javascripts/global.js just like for adding and deleting.

CONCLUSION

Well, there we have it. We've gotten through yet another big-ass NodeJS tutorial together. I hope everything worked!

This is obviously an incredibly rudimentary RESTful web app. It gives no thought to performance or to “what happens when my table has 1,000 users instead of 10?”-type questions. It also doesn't worry a lot about flexibility, maintainability, or future additions. The purpose was merely to get you up to speed with single-page, ajax-driven data manipulation in Node / Express / MongoDB. I hope it's accomplished that!

For more on REST and the best ways to approach it, I recommend the following articles:

- Best Practices for Designing a Pragmatic RESTful API
(<http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>) by Vinay Sahni
- Suggested REST API Practices (2013)
(<http://madhatted.com/2013/3/19/suggested-rest-api-practices>) by Matthew Beale

THANKS & INFO

I once again owe a debt of gratitude to a few people and would be remiss if I didn't give 'em some links!

- Azat Mardanov (https://twitter.com/azat_co), whose tutorial Node.js and MongoDB JSON REST API server with Mongoskin and Express.js (<http://webapplog.com/tutorial-node-js-and-mongodb-json-rest-api-server-with-mongoskin-and-express-js/>) helped me form some basic concepts for this one, and introduced me to mongoskin.
- Raquel Velez (<https://twitter.com/rockbot>) (who is now a part of the NPM team

(<https://npmjs.org/>)), who got me started on this whole Node thing.

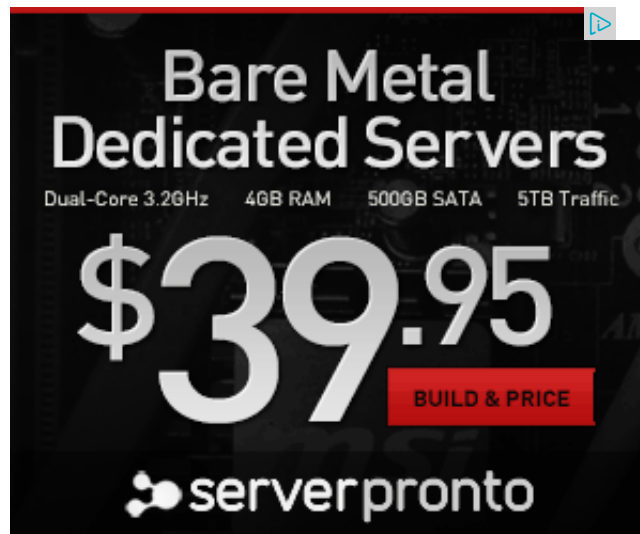
- The fine folks who helped proof this article:
 - Jordan Foreman
(<https://twitter.com/jordanforeman>)
 - Roger Keizer
(<https://twitter.com/RogerKeizer>)
- All the folks who commented on my previous tutorial. The response from across the web was amazing and really made me want to do more. I'm hoping to do video versions of that one (and this one) soon. Looking forward to hearing your thoughts on this tutorial!

This article was written in Markdown (<http://daringfireball.net/projects/markdown/>) over at Editorially (<http://editorially.com>) before being transferred to my HTML editor for final tweaking.

ABOUT THE AUTHOR

Christopher Buecheler (<http://cwbuecheler.com>) is an autodidact polymath, which is an *incredibly* pretentious way of saying that he's a jack of all trades who didn't like college. By day he's a front-end developer (<http://cwbuecheler.com/web/>) for a small San Francisco startup. By night he's a popular novelist (<http://cwbuecheler.com/writing/>), with four books released. He also is an award-winning amateur mixologist (http://www.huffingtonpost.com/tony-sachs/a-cocktail-safari-with-ta_b_867015.html) who writes cocktail articles for Primer Magazine (<http://cwbuecheler.com/writing/#nonfiction>) and runs a cocktail blog (<http://drinkshouts.com>), and he brews beer on occasion. He follows the NBA avidly and the NFL casually (and sometimes glances at MLB). He lives in Providence, Rhode Island, with his awesome French wife and their two cats. He is trying to learn French but wishes he could just download it from the Matrix.

ADVERTISEMENT



CURRENT PROJECTS

PULSE (/WRITING/#PULSE)

Version: 1st Draft

In Progress

Status: 60,000 Words

ELIXIR (/WRITING/#ELIXIR)

Version: Manuscript

Revising per agent's suggestions

Status: 104,200 Words

GET MY BOOKS

\$2.99

FREE

\$2.99

\$2.99

\$4.99

SUBSCRIBE TO MY NEWSLETTER

It's free, arrives bi-monthly, and contains no spam or ads.

SUBSCRIBE NOW
(/NEWSLETTER/)

FOLLOW ME ON TWITTER

Tweets

Follow

**DrinkShouts**

@DrinkShouts

7h

Tonight's [#cocktail](#): Old School Sazerac - cognac, sugar, absinthe, Peychaud's bitters. bit.ly/1s3dvhL | pic.twitter.com/8DS6bsHqie

Retweeted by Chris Buecheler

Show Photo

**Chris Buecheler**

@cwbuecheler

11h

The day may come when I manage to brew a batch of beer without getting sticky wort all over my kitchen floor ... BUT IT IS NOT THIS DAY!

Expand

**Chris Buecheler**

@cwbuecheler

14h

Also, I managed to write 950 words on Pulse while

Tweet to @cwbuecheler

Design • Code • Content © 2014 Christopher
Buecheler

SITE MENU

Home (/) Web (/web/) Writing (/writing/)

Cocktails (/cocktails/) Beer (/beer/)

[Video Games \(/videogames/\)](/videogames/) [Music \(/music/\)](/music/)

[Sports \(/sports/\)](/sports/) [Contact \(/contact/\)](/contact/)

BUILT WITH

HTML5 Boilerplate (<http://html5boilerplate.com/>)

CSS3 (<http://www.css3.info/>)

jQuery (<http://jquery.com/>)

SASS (<http://sass-lang.com/>)

Responsive Design
(<http://www.abookapart.com/products/responsive-web-design>)