

(/visit/heroku-151208-node.js)

How to Create a Complete Express.js + Node.js + MongoDB CRUD and REST Skeleton



Kendrick Coleman

Kendrick is a reformed sysadmin turned coder and awaits for an idea to spawn to tackle it with Ruby or Javascript. In his daily role a developer advocate for EMC Code, he works with a team to focus and publish all things open source.



Over the past few weeks I have created a few Node.js web apps (such as Bourbon Tweet Alerts (<http://bourbontweetalerts.cfapps.io>)) and I've noticed that there is a huge inconsistency in building Node.js web apps. I desperately needed a way to consistently build projects the same way every time. As we all know, there's nothing like having to figure out how someone made a piece of code function, much less an entire framework! I've been working with Ruby on Rails for a while, so making a transition into Node.js was quite a different leap. The one great thing I like about Rails is standardization. When you create a new rails app, the MVC architecture is templated out for you (ie "batteries included"). Node.js on the other hand, allows you to pick and choose any libraries you want to make the project successful. I've read over countless blogs and not a single one covers everything you need to know how to build a CRUD + REST application from soup to nuts. So I must thank about 20 different blogs and loads of github README's to help assemble this beacon of guidance.

I'll admit it, I'm a sucker for simple generation of skeleton. That's why I loved using rails. The entire MVC structure is laid out for me and I can be on my way.

```
rails new myapp --database=postgresql
rails generate scaffold Blob name:string badge:integer dob:date isloved:boolean
```

Generate the Express.js Skeleton

Within Node.js, the standard for creating web apps is Express.js (<http://expressjs.com/>). Express even has a template generator. Since you know I'm a sucker for it, let's use that. I'm going to assume you have Node.js & NPM already installed (<http://nodejs.org/download/>).

```
npm install -g express
npm install -g express-generator
```

Great. Now we've got Express & the generator installed as a global command on our system. To create a skeleton site go to your projects directory. When we create our application, a new folder with all the associated pieces are going to be created with it. By default this will use Jade (<http://jade-lang.com>) as the html templating engine. There are other templating engines such as EJS (<http://www.embeddedjs.com/>), Mustache (<https://mustache.github.io/>) and Hogan (<http://twitter.github.io/hogan.js/>), but once you start understanding Jade, it's very clean for simple web apps. Change out `nodewebapp` with whatever you want to call your application

```
express nodewebapp
```

Yep, it was that easy. You should now see these lines start scrolling:

```
create : nodewebapp
create : nodewebapp/package.json
create : nodewebapp/app.js
create : nodewebapp/public
create : nodewebapp/public/images
create : nodewebapp/public/stylesheets
create : nodewebapp/public/stylesheets/style.css
create : nodewebapp/routes
create : nodewebapp/routes/index.js
create : nodewebapp/routes/users.js
create : nodewebapp/public/javascripts
create : nodewebapp/views
create : nodewebapp/views/index.jade
create : nodewebapp/views/layout.jade
create : nodewebapp/views/error.jade
create : nodewebapp/bin
create : nodewebapp/bin/www

install dependencies:
$ cd nodewebapp && npm install
```

Setup MongoDB

Do you have MongoDB (<http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/>) setup? If you haven't, no worries. It's actually incredibly easy with your Mac and Homebrew (<http://brew.sh/>).

```
brew update
brew install mongodb
```

Let's start our MongoDB server. Side Note: I prefer to use iTerm2 (<http://iterm2.com/>) to have multiple tabs and windows with terminal prompts. From the terminal type `mongod` and you will see Mongo start up. By default, it will always use port 27017.

```
kcoleman-mbp: kcoleman$ mongod
mongod --help for help and startup options
2015-03-04T09:59:26.150-0500 [initandlisten] MongoDB starting : pid=13761 port=27017 dbpath=/dat
2015-03-04T09:59:26.150-0500 [initandlisten] db version v2.6.7
```

```

2015-03-04T09:59:26.150-0500 [initandlisten] git version: nogitversion
2015-03-04T09:59:26.150-0500 [initandlisten] build info: Darwin minimavericks.local 13.4.0 Darwi
2015-03-04T09:59:26.150-0500 [initandlisten] allocator: tcmalloc
2015-03-04T09:59:26.150-0500 [initandlisten] options: {}
2015-03-04T09:59:26.151-0500 [initandlisten] journal dir=/data/db/journal
2015-03-04T09:59:26.152-0500 [initandlisten] recover : no journal files present, no recovery nee
2015-03-04T09:59:26.184-0500 [initandlisten] waiting for connections on port 27017

```

From another terminal window, lets create a new database (<http://docs.mongodb.org/manual/tutorial/getting-started/>). We need to use the mongo client and 1 command to create it. Type `mongo` to enter the client/shell and use `dbname`. Of course change `dbname` to whatever name you want to call it. Here is the shell output.

```

kcoleman-mbp:kcoleman$ mongo
MongoDB shell version: 2.6.7
connecting to: test
> use nodewebappdb
switched to db nodewebappdb
> db
nodewebappdb
> show dbs
admin                (empty)
local                0.078GB
nodews4              0.078GB
nodewebappdb         0.078GB
>

```

Setup Mongoose to MongoDB

We will need to install the dependencies before we fire up the server, but I like to make sure we get our database pieces sorted first. Open up your favorite editor, such as Sublime Text (<http://www.sublimetext.com/>) or Atom (<https://atom.io/>), and make a new folder within nodewebapp called `model`. Inside this folder create a new file called `db.js`. This is the file where we are going to place our database connections.

```

1 | var mongoose = require('mongoose');                                javascript
2 | mongoose.connect('mongodb://localhost/nodewebappdb');

```

Next open up `app.js` and lets add our `db.js` file to the variables. by adding the line `var db = require('./model/db');`

```

1 | var express = require('express'),                                javascript
2 |   path = require('path'),
3 |   favicon = require('serve-favicon'),
4 |   logger = require('morgan'),
5 |   cookieParser = require('cookie-parser'),
6 |   bodyParser = require('body-parser'),
7 |   db = require('./model/db'),
8 |   routes = require('./routes/index'),
9 |   users = require('./routes/users');
10 | ..
11 | ..
12 | ..

```

Install Dependencies

Now we need to install our dependencies. from the terminal of the working directory of nodewebapp type

```
npm install
```

and you should see a big output

```

├─ raw-body@1.3.2
├─ depd@1.0.0
├─ qs@2.3.3
(/H) └─ iconv-lite@0.4.6
├─ on-finished@2.2.0 (ee-first@1.1.0)
├─ type-is@1.5.7 (mime-types@2.0.9)
(/dashboard)
├─ express@4.11.2 node_modules/express
├─ utils-merge@1.0.0
(/billing) └─ methods@1.1.1
├─ cookie@0.1.2
├─ fresh@0.2.4
├─ range-parser@1.0.2
├─ merge-descriptors@0.0.2
├─ cookie-signature@1.0.5
├─ escape-html@1.0.1
├─ vary@1.0.0
├─ media-typer@0.3.0
├─ parseurl@1.3.0
└─ finalhandler@0.3.3

```

We need to add a few more packages to our package.json file as a dependencies.

Mongoose will be our connection to MongoDB, body-parser is used to examine POST calls, and method-override is used by express to create DELETE and PUT requests through forms. read about method-override (<https://github.com/expressjs/method-override>)

```

npm install mongoose --save
npm install body-parser --save
npm install method-override --save

```

Awesome. Now to test it's all working in the terminal type `npm start` and you shouldn't see any errors:

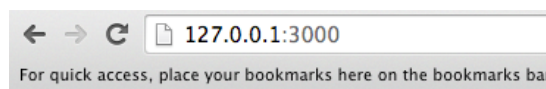
```

kcoleman-mbp:nodewebapp kcoleman$ npm start

> nodewebapp@0.0.1 start /Users/kcoleman/Documents/projects/nodewebapp
> node ./bin/www

```

and if you navigate to `http://127.0.0.1:3000` you will see the following screen:



Express

Welcome to Express

HOORAY!

We've got a functioning web server that is talking to Mongo. Part 1 is accomplished.

Coming from a Rails background, it was pretty easy to create a new model with all the routes, controllers and views through a simple scaffold. Node.js on the other hand requires most of this to be done manually.

Create the Model and Schema

Remember that `model` folder we created earlier? We are going to use that folder to form our objects. Create a new file within that folder for whatever your object is going to be. In this case, I'm going to call it `blobs.js`. Use the plural form of the object.

Each `blob` is going to have a Name (string), Badge (number), DOB (date), and IsLoved (boolean). Mongoose Schema Types (<http://mongoosejs.com/docs/guide.html>)

```
1 | var mongoose = require('mongoose');                                javascript
2 | var blobSchema = new mongoose.Schema({
3 |   name: String,
4 |   badge: Number,
5 |   dob: { type: Date, default: Date.now },
6 |   isloved: Boolean
7 | });
8 | mongoose.model('Blob', blobSchema);
```

Within `app.js` add this to your variables at the top below our `db` variable we added earlier

```
...
db = require('./model/db'),
blob = require('./model/blobs'),
...
```

Create the Controller

We've now come to a crossroad. Many blogs and tutorials will show the following using the `index.js` file. IMO, that's bad business. Let's leave `index.js` for front page stuff if you want to add it later on like Contact, About, Login, etc. Even if you are making a simple single-page web app, we can change `app.js` to use a different controller for `/`.

Create a new file in the `routes` folder. In this example, I'm going to create a new file called `blobs.js`.

Here's the fun part! We're going to build our entire controller with all the CRUD and REST pieces completely baked in. We are going to take this piece by piece but all of this will go into the `blob.js` file.

Define the packages we will need

```
1 | var express = require('express'),                                javascript
2 |   router = express.Router(),
3 |   mongoose = require('mongoose'), //mongo connection
4 |   bodyParser = require('body-parser'), //parses information from POST
5 |   methodOverride = require('method-override'); //used to manipulate POST
```

This portion must be placed before we get to our CRUD and REST. This is completely copy and pasted from `method-override` (<https://github.com/expressjs/method-override>). Using `use` will make sure that every requests that hits this controller will pass through these functions.

```
1 | router.use(bodyParser.urlencoded({ extended: true })))            javascript
2 | router.use(methodOverride(function(req, res){
3 |   if (req.body && typeof req.body === 'object' && '_method' in req.body) {
```

```

4      // look in urlencoded POST bodies and delete it
5      var method = req.body._method
6      delete req.body._method
7      return method
8    }
9  })

```

Our first big part of the CRUD. We are going to build the GET for grabbing all the Blobs from the database to display it and we are also going to build the POST for creating a new blob.

```

1 //build the REST operations at the base for blobs                                javascript
2 //this will be accessible from http://127.0.0.1:3000/blobs if the default route for / is
3 router.route('/')
4   //GET all blobs
5   .get(function(req, res, next) {
6     //retrieve all blobs from Monogo
7     mongoose.model('Blob').find({}, function (err, blobs) {
8       if (err) {
9         return console.error(err);
10      } else {
11        //respond to both HTML and JSON. JSON responses require 'Accept: applic
12        res.format({
13          //HTML response will render the index.jade file in the views/blobs
14          html: function(){
15            res.render('blobs/index', {
16              title: 'All my Blobs',
17              "blobs" : blobs
18            });
19          },

```

We don't want to spend all of our time creating new Blobs through REST calls, so we are going to use this bit of code to wire up a page called `new.jade` that will display a form. We will build the form a bit later when we get to the Views.

```

1 /* GET New Blob page. */                                                         javascript
2 router.get('/new', function(req, res) {
3   res.render('blobs/new', { title: 'Add New Blob' });
4 });

```

This middleware will be used to do error checking. Basically, making sure that we have legit entries in the database for the `:id`.

```

1 // route middleware to validate :id                                             javascript
2 router.param('id', function(req, res, next, id) {
3   //console.log('validating ' + id + ' exists');
4   //find the ID in the Database
5   mongoose.model('Blob').findById(id, function (err, blob) {
6     //if it isn't found, we are going to repond with 404
7     if (err) {
8       console.log(id + ' was not found');
9       res.status(404)
10      var err = new Error('Not Found');
11      err.status = 404;
12      res.format({
13        html: function(){

```

```

14         next(err);
15     },
16     json: function(){
17         res.json({message : err.status + ' ' + err});
18     }
19 });

```

Now we need to GET an individual blob to display it. We will wire this up later with an `show.jade` form. This example shows how to grab it by ID.

```

1 router.route('/:id')
2   .get(function(req, res) {
3     mongoose.model('Blob').findById(req.id, function (err, blob) {
4       if (err) {
5         console.log('GET Error: There was a problem retrieving: ' + err);
6       } else {
7         console.log('GET Retrieving ID: ' + blob._id);
8         var blobdob = blob.dob.toISOString();
9         blobdob = blobdob.substring(0, blobdob.indexOf('T'))
10         res.format({
11           html: function(){
12             res.render('blobs/show', {
13               "blobdob" : blobdob,
14               "blob" : blob
15             });
16           },
17           json: function(){
18             res.json(blob);
19           }

```

Next, we need a way to edit and update our document through a standard web form. We will wire this up with `edit.jade` a bit later on (NOTE: please review the final code at [express-node-mongo-skeleton](https://github.com/kacole2/express-node-mongo-skeleton) on GitHub (<https://github.com/kacole2/express-node-mongo-skeleton>) to see how GET, PUT, and DELETE can be combined for the `/:id/edit` pieces.)

```

1 //GET the individual blob by Mongo ID
2 router.get('/:id/edit', function(req, res) {
3   //search for the blob within Mongo
4   mongoose.model('Blob').findById(req.id, function (err, blob) {
5     if (err) {
6       console.log('GET Error: There was a problem retrieving: ' + err);
7     } else {
8       //Return the blob
9       console.log('GET Retrieving ID: ' + blob._id);
10      //format the date properly for the value to show correctly in our edit form
11      var blobdob = blob.dob.toISOString();
12      blobdob = blobdob.substring(0, blobdob.indexOf('T'))
13      res.format({
14        //HTML response will render the 'edit.jade' template
15        html: function(){
16          res.render('blobs/edit', {
17            title: 'Blob' + blob._id,
18            "blobdob" : blobdob,
19            "blob" : blob

```

PUT is used to update the blob in case we need to make changes. This put accepts REST and form POST requests. This looks eerily similar to POST from earlier because we are basically doing the same thing. Take values, assign values, and then update values.

```

1 //PUT to update a blob by ID
2 router.put('/:id/edit', function(req, res) {
3   // Get our REST or form values. These rely on the "name" attributes
4   var name = req.body.name;
5   var badge = req.body.badge;
6   var dob = req.body.dob;
7   var company = req.body.company;
8   var isloved = req.body.isloved;

```

```

9
10 //find the document by ID
11 mongoose.model('Blob').findById(req.id, function (err, blob) {
12     //update it
13     blob.update({
14         name : name,
15         badge : badge,
16         dob : dob,
17         isloved : isloved
18     }, function (err, blobID) {
19         if (err) {

```

DELETE is a pretty crucial part of CRUD. As usual, this will be accepted by REST and POSTs from a form

```

1 //DELETE a Blob by ID
2 router.delete('/:id/edit', function (req, res){
3     //find blob by ID
4     mongoose.model('Blob').findById(req.id, function (err, blob) {
5         if (err) {
6             return console.error(err);
7         } else {
8             //remove it from Mongo
9             blob.remove(function (err, blob) {
10                 if (err) {
11                     return console.error(err);
12                 } else {
13                     //Returning success messages saying it was deleted
14                     console.log('DELETE removing ID: ' + blob._id);
15                     res.format({
16                         //HTML returns us back to the main page, or you can create a succ
17                         html: function(){
18                             res.redirect("/blobs");
19                     }

```

Export all of our routes.

```
1 module.exports = router;
```

That's IT! We have built a completely functional controller that contains the ability to do all CRUD operations through web pages as well as REST calls that will respond with JSON formatting. pretty awesome!

Add The Route

`app.js` is the brains for understanding how to get to everything in your app. We need to add our blobs route variable and add our blob route below the main `/` route. This means that our Blob model will be accessed from `http://127.0.0.1:3000/blobs`

```

var routes = require('./routes/index');
var blobs = require('./routes/blobs');
```

```

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

Are you building a single-page web app? you can always change the root to point to blobs by using `app.use('/', blobs);`. Then our blobs are accessed at `http://127.0.0.1:3000`. This may require you to go back to your controller and change some `res.redirect` pieces.

Add The Views

ADD THE VIEWS

Now we need to create our views. Create a new folder inside of `views`. In my case, I'm going to call it `blobs`. Reserve the jade files in the root of `views` for something later. We are going to create views for `index`, `new`, and `edit`

All of these views are using `layout.jade` as a building block. Since this layout lives in a folder one directory up, we preface it with `../`. The result will be `extends ../layout`. If you don't, then you will encounter an error that the app can't find `layout.jade`.

Create a new `index.jade` file within the `blobs` folder. This index file is what is used to show all blobs. Yes, this is ugly but it will atleast give us a look at how it all functions. The `form` allows us to delete the document via a button using a POST operation. The Edit link will take us to another page using GET where we can edit information

```

extends ../layout

block content
  h1.
    #{title}
  ul
    - each blob, i in blobs
      li
        = blob.name
        = blob.badge
        = blob.dob
        = blob.isloved
        = blob._id
        form(action='/blobs/#{blob._id}/edit',method='post',enctype='application/x-www
          input(type='hidden',value='DELETE',name='_method')
          button(type='submit').
            Delete
        p
          a(href='/blobs/#{blob._id}/edit') Edit

```

127.0.0.1:3000/blobs

Access, place your bookmarks here on the bookmarks bar. [Import bookmarks now...](#)

All my Blobs

- Kendrick Coleman123456Sat May 07 1983 20:00:00 GMT-0400 (EDT>true550853772b3033c92dc6f8ab

[Edit](#)
[Show](#)

As you can probably guess, we still need a way to add Blobs via a form. Create a new file called `new.jade`. This page is accessible from `/blobs/new`.

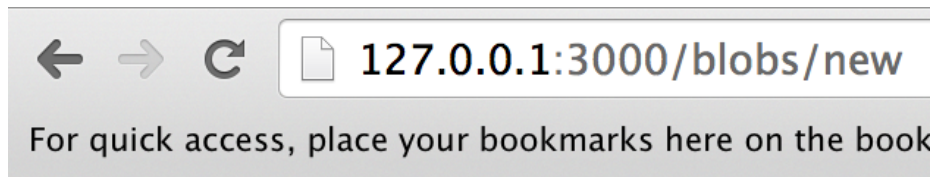
```

extends ../layout

block content
  h1.
    #{title}
  form#formAddBlob(name="addblob",method="post",action="/blobs")
    p Name:
      input#inputName(type="text", placeholder="ex. John Smith", name="name")
    p Badge:
      input#inputBadge(type="number", placeholder="ex. 123456", name="badge")
    p DOB:
      input#inputDob(type="date", name="dob")
    p Are You Loved?:
      input#inputIsLoved(type="checkbox", name="isloved")
    p

```

```
button#btnSubmit(type="submit") submit
```



Add New Blob

Name:

Badge:

DOB:

Are You Loved?: ☐

To show an individual blob we need to use `show.jade`. This page is accessible from `/blobs/:id`.

```
extends ../layout
```

```
block content
```

```
h1.
```

```
  Infophoto #{blob._id}
```

```
  p Name: #{blob.name}
```

```
  p Badge: #{blob.badge}
```

```
  p DOB: #{blobdob}
```

```
  p Is Loved: #{blob.isloved}
```



Infophoto 550853772b3033c92dc6f8ab

Name: Kendrick Coleman

Name: Kendrick Coleman

Badge: 123456

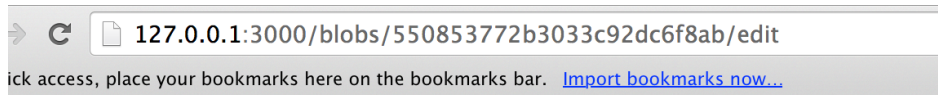
DOB: 1983-05-08

Is Loved: true

Lastly, we need a form that shows the pre-populated data and allows us to manipulate it and update it via PUT. Create a new file called `edit.jade` that will be accessible by `/blobs/:id/edit`

```
extends ../layout

block content
  h1.
    Blob ID #{blob._id}
  form(action='/blobs/#{blob._id}/edit',method='post',name='updateblob',enctype='application/x-www-form-urlencoded')
    p Name:
      input#inputName(type='text', value='#{blob.name}', name='name')
    p Badge:
      input#inputBadge(type='number', value='#{blob.badge}', name='badge')
    p DOB:
      input#inputDob(type='date', value='#{blobdob}', name='dob')
    p Are you Loved?:
      input#inputIsLoved(type='checkbox', name='isloved', checked=('#{blob.isloved}'==='true'))
    p
      input(type='hidden',value='PUT',name='_method')
    p
      button#btnSubmit(type='submit').
        Update
```



Blob ID 550853772b3033c92dc6f8ab

Name:

Badge:

DOB:

Are you Loved?: ☒

##You Did It! Whew... that's it! Congrats! You've officially created an entire CRUD

application with all the usual REST calls baked in. Now you can fire up your server with `npm start` and begin adding new Blobs!

Here is what your folder structure should look like:

```
nodewebapp
├── README.md
├── app.js
├── package.json
├── bin
│   └── www
├── model
│   ├── blobs.js
│   └── db.js
├── node_modules
│   └── too many to list
│       └── ...
└── public
    ├── images
    ├── javascripts
    └── stylesheets
```

Is something not working correctly? Check out my [express-node-mongo-skeleton](https://github.com/kacole2/express-node-mongo-skeleton) (<https://github.com/kacole2/express-node-mongo-skeleton>) so you can see everything in its entirety.

How would you rate the quality of this post?

3 stars (average) ▼

Why did you give this post 3 stars?

SAVE REVIEW

Reviews (2)

Rating 4.5/5



TMin 11 days ago

Great Tutorial. Only problem is `http://127.0.0.1:3000/blobs` doesn't load. it just keeps trying to connect. `npm start` seems compiles and runs. All in all a great tutorial though. Thanks

Upvote - Reply



Charles Johnston 6 days ago

try `localhost:3000/blobs` instead.



P. Ross Baldwin 10 days ago

Very good, but somewhat hard to understand. Most likely because i am a beginner.

Upvote - Reply



Deploy Node.js apps with focus and flow

Platform features you need, developer experience you'll love.

[SIGN UP FOR FREE](#)

(/visit/heroku-151208-node.js)

Tagged under

- javascript (/posts/tag/javascript)
- node.js (/posts/tag/node.js)
- mongodb (/posts/tag/mongodb)
- rest (/posts/tag/rest)
- crud (/posts/tag/crud)

Similar posts

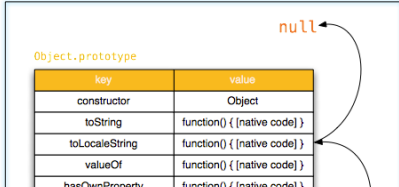
ALL POSTS (/SOFTWARE-EXPERTS)



Tips for Dealing with Developer Stress

Rich McLaughlin


<https://www.airpair.com/javascript/posts/tips-for-dealing-with-developer-stress>



How does JavaScript .prototype work?

Mehran Hatami

<https://www.airpair.com/javascript/posts/how-does-javascript-prototype-work>



Mastering ES6 higher-order functions for Arrays

Tiago Romero Garcia

<https://www.airpair.com/javascript/posts/mastering-es6-higher-order-functions-for-arrays>