SPA, todo App w/ node & angular 03\_2013.11.07; dev; notes Part 1~3

# **Creating a Single Page Todo App with Node and Angular 1/3 @@**[**http://scotch.io/tutorials/javascript/creating-a-single-page-todo-app-with-node-and-angular**](http://scotch.io/tutorials/javascript/creating-a-single-page-todo-app-with-node-and-angular)

# **Organization & Structure 2/3 @@http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-application-organization-and-structure**

# **Controllers & Service (modularized) 3/3 @@http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-controllers-and-services**

*·*[*angularJS*](http://scotch.io/tag/angular-js)*,* [*MEAN*](http://scotch.io/tag/mean)*,* [*node.js*](http://scotch.io/tag/node-js) [*Chris Sevilleja*](http://scotch.io/tutorials/javascript/creating-a-single-page-todo-app-with-node-and-angular)  *November 7, 2013* [*68 Comments*](http://scotch.io/tutorials/javascript/creating-a-single-page-todo-app-with-node-and-angular#comment-section)

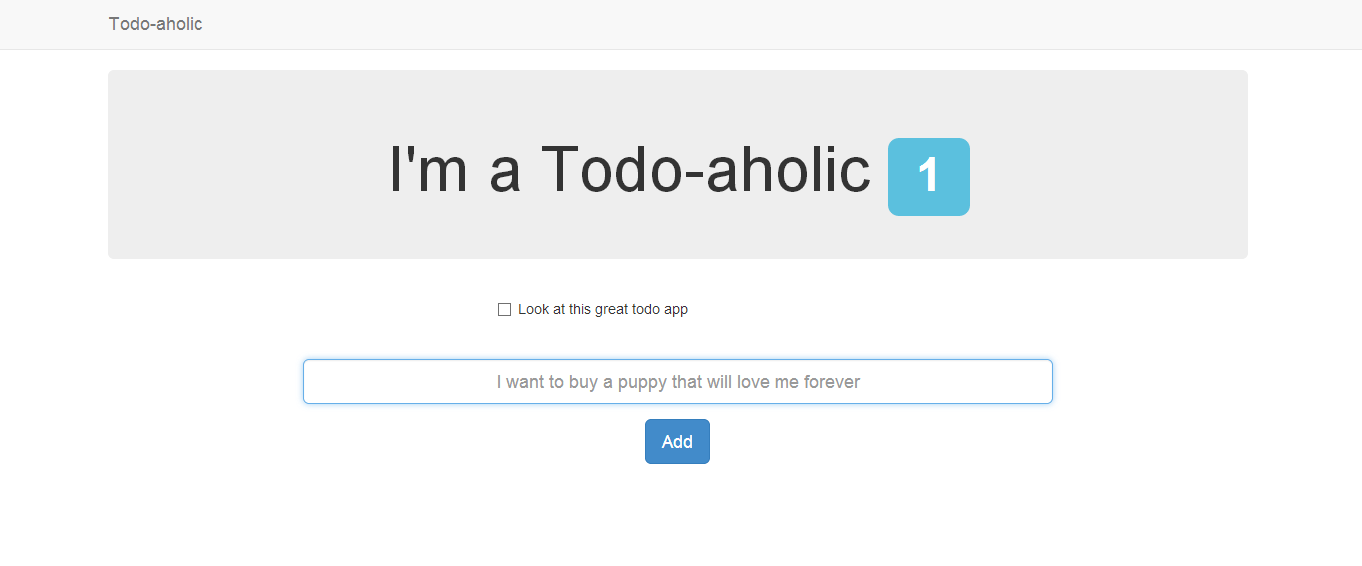
# **Single Page Application with Node and Angular**

Today we will be creating a very simple Todo application using the MEAN (Mongo, Express, Angular, Node) stack. We will be creating:

* Single page application to create and finish todos
* Storing todos in a MongoDB using Mongoose
* Using the Express framework
* Creating a RESTful Node API
* Using Angular for the frontend and to access the API

While the application is simple and **beginner to intermediate** level in its own right, the concepts here can apply to much more advanced apps. The biggest things we should focus on is using Node as an API and Angular as the frontend. Making them work together can be a bit confusing so this tutorial should help alleviate some confusion. Buckle those seatbelts; this could be a long one.

### What We’ll Be Building



## Base Setup

### File Structure

We are going to keep the file structure very simple and put most of the code for our Node application into the server.js file. In larger applications, this should be broken down further to separate duties. [Mean.io](http://mean.io/) is a good boilerplate to see best practices and how to separate file structure. Let’s go ahead and create our simpler file structure and edit the files as we go along.

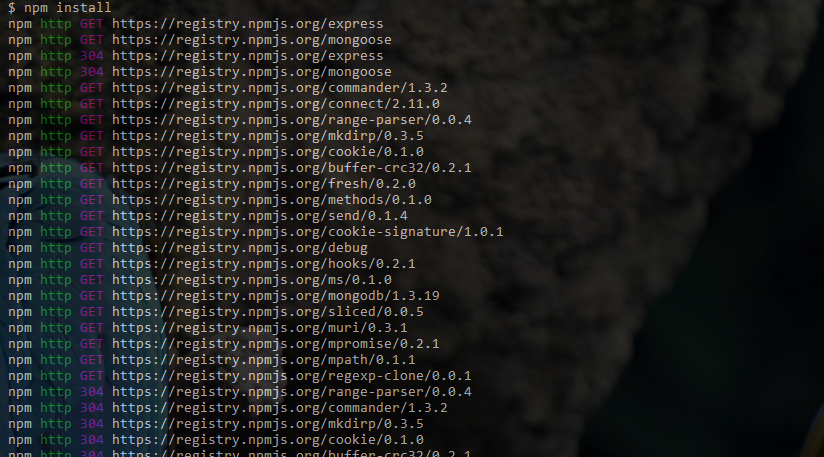
- public <!-- holds all our files for our frontend angular application -->  
 ----- core.js <!-- all angular code for our app -->  
 ----- index.html <!-- main view -->  
 - package.json <!-- npm configuration to install dependencies/modules -->  
 - server.js <!-- Node configuration -->

### Installing Modules

In Node, the package.json file holds the configuration for our app. Node’s package manager (npm) will use this to install any dependencies or modules that we are going to use. In our case, we will be using [Express](http://expressjs.com/) (popular Node framework) and [Mongoose](http://mongoosejs.com/) (object modeling for MongoDB).

// package.json  
 {  
 "name" : "node-todo",  
 "version" : "0.0.0",  
 "description" : "Simple todo application.",  
 "main" : "server.js",  
 "author" : "Scotch",  
 "dependencies" : {  
 "express" : "~3.4.4",  
 "mongoose" : "~3.6.2"  
 }  
 }

Now if we run npm install, npm will look at this file and install Express and Mongoose.



### Node Configuration

In our package.json file, we told it that our main file would be server.js. This is the main file for our Node app and where we will configure the entire application.

This is the file where we will:

* Configure our application
* Connect to our database
* Create our Mongoose models
* Define routes for our RESTful API
* Define routes for our frontend Angular application
* Set the app to listen on a port so we can view it in our browser

For now, we will just configure the app for Express, our MongoDB database, and listening on a port.

// server.js  
  
 // set up ========================  
 var express = require('express');  
 var app = express(); // create our app w/ express  
 var mongoose = require('mongoose'); // mongoose for mongodb  
  
 // configuration =================  
  
 mongoose.connect('mongodb://node:node@mongo.onmodulus.net:27017/uwO3mypu'); // connect to mongoDB database on modulus.io  
  
 app.configure(function() {  
 app.use(express.static(\_\_dirname + '/public')); // set the static files location /public/img will be /img for users  
 app.use(express.logger('dev')); // log every request to the console  
 app.use(express.bodyParser()); // pull information from html in POST  
 app.use(express.methodOverride()); // simulate DELETE and PUT  
 });  
  
 // listen (start app with node server.js) ======================================  
 app.listen(8080);  
 console.log("App listening on port 8080");

Just with that bit of code, we now have an HTTP server courtesy of Node. We have also created an app with Express and now have access to many benefits of it. In our app.configure section, we are using express modules to add more functionality to our application.

### Database Setup

We will be using a remote database hosted on [Modulus.io](https://modulus.io/). They provide a great service and give you $15 upfront to use as you see fit. This is great for doing testing and creating databases on the fly.

Modulus will provide the database URL you need and you can use mongoose.connect to connect to it. That’s it.

### Start Your App!

Now that we have our package.json and server.js started up, we can start up our server and see what’s going on. Just go into your console and use the following command:

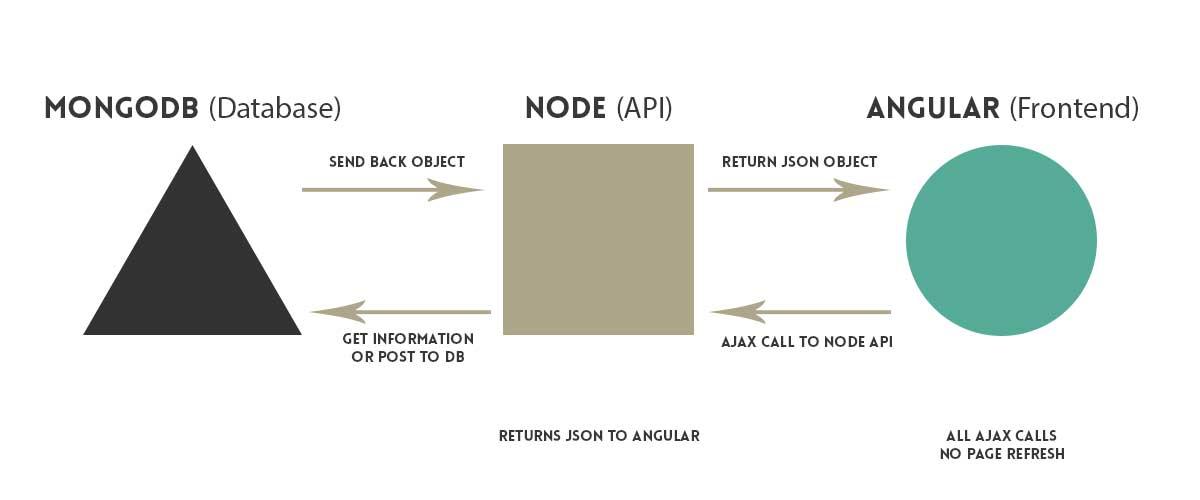
node server.js

Now you have a server listening on port 8080. You can’t see anything in your browser at **http://localhost:8080**yet since we didn’t configure our application to output anything. But it’s a start!

**Automatically restart server when files change:** By default, node will not monitor for file changes after your server has been started. This means you’d have to shut down and start the server every time you made a file change. This can be fixed with **nodemon**. To use: install nodemon globally npm install -g nodemon. Start your server with nodemon server.js now. Smooth sailing from there.

## Application Flow

Now a brief overview of how all our moving parts will work together. There are a lot of different ideas and technologies involved in this application that it is easy to get mixed up with them all. In our diagram below, we explain a bit of the separation of tasks and how the parts tie in together.



Angular is on its own in the frontend. It accesses all the data it needs through the Node API. Node hits the database and returns JSON information to Angular based on the RESTful routing.

This way, you can separate the frontend application from the actual API. If you want to extend the API, you can always build more routes and functions into it without affecting the frontend Angular application. This way you can eventually build different apps on different platforms since you just have to hit the API.

## Creating Our Node API

Before we get to the frontend application, we need to create our RESTful API. This will allow us to have an api that will **get all todos**, **create a todo**, and **complete and delete a todo**. It will return all this information in JSON format.

### Todo Model

We must define our model for our Todos. We’ll keep this simple. After the **configuration section** and before the **listen** section, we’ll add our model.

// define model =================  
 var Todo = mongoose.model('Todo', {  
 text : String  
 });

That is all we want. Just the text for the todo. MongoDB will automatically generate an \_id for each todo that we create also.

### RESTful API Routes

Let’s generate our Express routes to handle our API calls.

// server.js  
...  
  
// routes ======================================================================  
  
 // api ---------------------------------------------------------------------  
 // get all todos  
 app.get('/api/todos', function(req, res) {  
  
 // use mongoose to get all todos in the database  
 Todo.find(function(err, todos) {  
  
 // if there is an error retrieving, send the error. nothing after res.send(err) will execute  
 if (err)  
 res.send(err)  
  
 res.json(todos); // return all todos in JSON format  
 });  
 });  
  
 // create todo and send back all todos after creation  
 app.post('/api/todos', function(req, res) {  
  
 // create a todo, information comes from AJAX request from Angular  
 Todo.create({  
 text : req.body.text,  
 done : false  
 }, function(err, todo) {  
 if (err)  
 res.send(err);  
  
 // get and return all the todos after you create another  
 Todo.find(function(err, todos) {  
 if (err)  
 res.send(err)  
 res.json(todos);  
 });  
 });  
  
 });  
  
 // delete a todo  
 app.delete('/api/todos/:todo\_id', function(req, res) {  
 Todo.remove({  
 \_id : req.params.todo\_id  
 }, function(err, todo) {  
 if (err)  
 res.send(err);  
  
 // get and return all the todos after you create another  
 Todo.find(function(err, todos) {  
 if (err)  
 res.send(err)  
 res.json(todos);  
 });  
 });  
 });  
  
...

Based on these routes, we’ve built a table to explain how a frontend application should request data from the API.

|  |  |  |
| --- | --- | --- |
| HTTP Verb | URL | Description |
| GET | /api/todos | Get all of the todos |
| POST | /api/todos | Create a single todo |
| DELETE | /api/todos/:todo\_id | Delete a single todo |

Inside of each of our API routes, we use the Mongoose actions to help us interact with our database. We created our Model earlier with var Todo = mongoose.model and now we can use that to **find**, **create**, and **remove**. There are many more things you can do and I would suggest looking at the official [docs](http://mongoosejs.com/docs/guide.html) to learn more.

Our API is done! Rejoice! If you start up your application, you can interact with it at localhost:8080/api/todos to get all the todos. There won’t be anything currently since you haven’t added any.

## Frontend Application with Angular

We have **created a Node application**, **configured our database**, **generated our API routes**, and **started a server**. So much already done and still a little bit longer to go!

The work that we’ve done so far can stand on its own as an application. It can be an API we use let applications and users connect with our content.

We want to be the first to use our brand new API that we’ve just created. This is one of my favorite terms that I learned about last month: We will be [dogfooding](http://en.wikipedia.org/wiki/Eating_your_own_dog_food). We could treat this as we are our very first client to use our new API. We are going to keep this simple so we’ll have just our index.html and core.js to define our frontend.

### Defining Frontend Route

We have already defined our API routes. Our application’s API is accessible from /api/todos, but what about our frontend? How do we display the index.html file at our home page?

We will add one route to our server.js file for the frontend application. This is all we need to do since Angular will be making a single page application and handle the routing.

After our API routes, and before the app.listen, add this route:

// server.js  
...  
 // application -------------------------------------------------------------  
 app.get('\*', function(req, res) {  
 res.sendfile('./public/index.html'); // load the single view file (angular will handle the page changes on the front-end)  
 });  
...

This will load our single index.html file when we hit localhost:8080.

### Setting Up Angular core.js

Let’s go through our Angular setup first. We have to **create a module**, **create a controller**, and **define functions to handle todos**. Then we can **apply to view**.

// public/core.js  
var scotchTodo = angular.module('scotchTodo', []);  
  
function mainController($scope, $http) {  
 $scope.formData = {};  
  
 // when landing on the page, get all todos and show them  
 $http.get('/api/todos')  
 .success(function(data) {  
 $scope.todos = data;  
 console.log(data);  
 })  
 .error(function(data) {  
 console.log('Error: ' + data);  
 });  
  
 // when submitting the add form, send the text to the node API  
 $scope.createTodo = function() {  
 $http.post('/api/todos', $scope.formData)  
 .success(function(data) {  
 $scope.formData = {}; // clear the form so our user is ready to enter another  
 $scope.todos = data;  
 console.log(data);  
 })  
 .error(function(data) {  
 console.log('Error: ' + data);  
 });  
 };  
  
 // delete a todo after checking it  
 $scope.deleteTodo = function(id) {  
 $http.delete('/api/todos/' + id)  
 .success(function(data) {  
 $scope.todos = data;  
 console.log(data);  
 })  
 .error(function(data) {  
 console.log('Error: ' + data);  
 });  
 };  
  
}

We create our Angular module (scotchApp) and controller (mainController).

We also create our functions to **get all todos**, **create a todo**, and **delete a todo**. All these will be hitting the API we just created. On page load, we will GET /api/todos and bind the JSON we receive from the API to$scope.todos. We will then loop over these in our view to make our todos.

We’ll follow a similar pattern for **creating and deleting**. Run our action, remake our todos list.

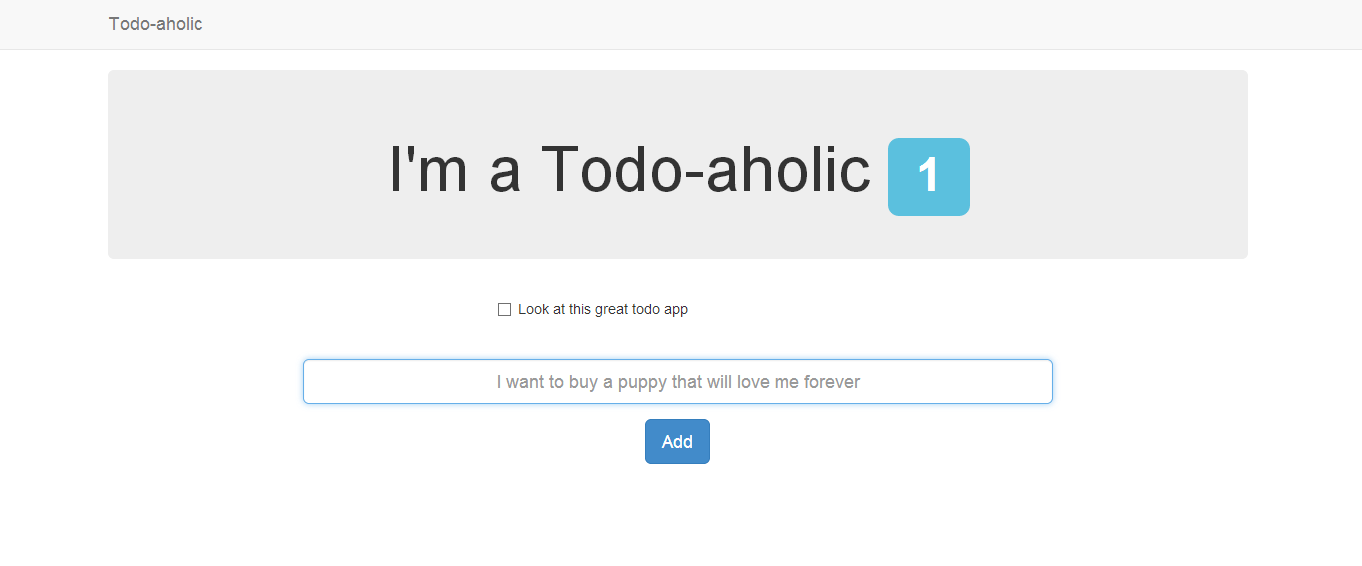
### Frontend View index.html

Here we will keep it simple. This is the HTML needed to interact with Angular. We will:

* Assign Angular module and controller
* Initialize the page by getting all todos
* Loop over the todos
* Have a form to create todos
* Delete todos when they are checked

<!-- index.html -->  
<!doctype html>  
  
<!-- ASSIGN OUR ANGULAR MODULE -->  
<html ng-app="scotchTodo">  
<head>  
 <!-- META -->  
 <meta charset="utf-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1"><!-- Optimize mobile viewport -->  
  
 <title>Node/Angular Todo App</title>  
  
 <!-- SCROLLS -->  
 <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css"><!-- load bootstrap -->  
 <style>  
 html { overflow-y:scroll; }  
 body { padding-top:50px; }  
 #todo-list { margin-bottom:30px; }  
 </style>  
  
 <!-- SPELLS -->  
 <script src="//ajax.googleapis.com/ajax/libs/jquery/2.0.3/jquery.min.js"></script><!-- load jquery -->  
 <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.0.8/angular.min.js"></script><!-- load angular -->  
 <script src="core.js"></script>  
  
</head>  
<!-- SET THE CONTROLLER AND GET ALL TODOS -->  
<body ng-controller="mainController">  
 <div class="container">  
  
 <!-- HEADER AND TODO COUNT -->  
 <div class="jumbotron text-center">  
 <h1>I'm a Todo-aholic <span class="label label-info">{{ todos.length }}</span></h1>  
 </div>  
  
 <!-- TODO LIST -->  
 <div id="todo-list" class="row">  
 <div class="col-sm-4 col-sm-offset-4">  
  
 <!-- LOOP OVER THE TODOS IN $scope.todos -->  
 <div class="checkbox" ng-repeat="todo in todos">  
 <label>  
 <input type="checkbox" ng-click="deleteTodo(todo.\_id)"> {{ todo.text }}  
 </label>  
 </div>  
  
 </div>  
 </div>  
  
 <!-- FORM TO CREATE TODOS -->  
 <div id="todo-form" class="row">  
 <div class="col-sm-8 col-sm-offset-2 text-center">  
 <form>  
 <div class="form-group">  
  
 <!-- BIND THIS VALUE TO formData.text IN ANGULAR -->  
 <input type="text" class="form-control input-lg text-center" placeholder="I want to buy a puppy that will love me forever" ng-model="formData.text">  
 </div>  
  
 <!-- createToDo() WILL CREATE NEW TODOS -->  
 <button type="submit" class="btn btn-primary btn-lg" ng-click="createTodo()">Add</button>  
 </form>  
 </div>  
 </div>  
  
 </div>  
  
</body>  
</html>

Take a look at what we have.



## Conclusion

Now we have a fully working application that will show, create, and delete todos all via API (that we built!). That was quite a day. We’ve done so much. Just an overview of what we’ve accomplished:

* RESTful Node API using Express
* MongoDB interaction using mongoose
* Angular AJAX $http calls
* Single page application w/ no refreshes
* Dogfooding (sorry, I really like that word)

### Test the Application

Go ahead and download the code on [Github](https://github.com/scotch-io/node-todo) and tweak it or test it. To get it all up and running:

1. Make sure you have [Node and npm](http://nodejs.org/) installed
2. Clone the repo: git clone git@github.com:scotch-io/node-todo
3. Install the application: npm install
4. Start the server: node server.js
5. View in your browser at http://localhost:8080

I hope this was insightful on how to have lots of moving parts work together. In the future, we will look at separating our server.js file since that got a little crazy.

**Edit #1:** Removing ng-init

This article is part of our [**Node and Angular To-Do App**](http://scotch.io/series/node-and-angular-to-do-app) series.

1. Creating a Single Page To-do App with Node and Angular
2. [**Node Application Organization and Structure**](http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-application-organization-and-structure)
3. [**Angular Modules: Controllers and Services**](http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-controllers-and-services)

# **Node and Angular To-Do App: Application Organization and Structure 2/3**

*·*[*angularJS*](http://scotch.io/tag/angular-js)*,* [*MEAN*](http://scotch.io/tag/mean)*,* [*node.js*](http://scotch.io/tag/node-js) [*Chris Sevilleja*](http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-application-organization-and-structure)  *November 13, 2013* [*13 Comments*](http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-application-organization-and-structure#comment-section)

This article will look into best practices for laying out and organizing a Node and Angular (MEAN stack) app.

In the process of extending our Node and Angular To-Do App with authorization, I ran into a problem: the tutorial was going to be huge. In our first tutorial, we placed everything (variables, configuration, models, routes, and more) in our server.js file. While this is fine for demonstration purposes and small Node applications, it wasn’t going to work with how far we wanted to take this To-Do app.

**Demo:** The demo will be the same as the first demo. The underlying code will be different however.

## Application Structure

The structure of the files in our application is very important. It can help us expand and grow the app as we will inevitably get more and more popular (hopefully), or it could make coming back to the code an absolute nightmare.

There are so many different ways to lay out a MEAN (MongoDB, Express, Angular, Node) app, and the tutorials online show that. This tries to lean close to best practices and the best ideas taken from many of those MEAN stack tutorials.

In this article, we’ll look at how we can separate the core functions of our app into a sensible and sane file structure. Let’s begin.

## Our Current Application

Since this is an extension of our first article, we will be using the same code. If you haven’t done so already, [download](https://github.com/scotch-io/node-todo/archive/v1.0.zip) that code so you can follow along, or if you have a ton of RAM in that brain of yours (or good memory), just read on through.

The code from the first article isn’t really necessary if you don’t want to pull it down. The concepts still apply to any MEAN application.

Our application **stores all functionality in server.js**. Here is our current file structure:

<!-- old file structure -->  
  
 - public <!-- holds all our files for our frontend angular application -->  
 ----- core.js <!-- all angular code for our app -->  
 ----- index.html <!-- main view -->  
 - package.json <!-- npm configuration to install dependencies/modules -->  
 - server.js <!-- Node configuration -->

Here is our server.js. Super long and could get confusing to read through in the future.

<!-- server.js -->  
  
// set up ======================================================================  
var express = require('express');  
var app = express(); // create our app w/ express  
var mongoose = require('mongoose'); // mongoose for mongodb  
  
var port = process.env.PORT || 8080;  
  
// configuration ===============================================================  
  
mongoose.connect('mongodb://node:node@mongo.onmodulus.net:27017/uwO3mypu'); // connect to mongoDB database on modulus.io  
  
app.configure(function() {  
 app.use(express.static(\_\_dirname + '/public')); // set the static files location /public/img will be /img for users  
 app.use(express.logger('dev')); // log every request to the console  
 app.use(express.bodyParser()); // pull information from html in POST  
 app.use(express.methodOverride()); // simulate DELETE and PUT  
});  
  
// define model ================================================================  
var Todo = mongoose.model('Todo', {  
 text : String,  
 done : Boolean  
});  
  
// routes ======================================================================  
  
 // api ---------------------------------------------------------------------  
 // get all todos  
 app.get('/api/todos', function(req, res) {  
 ...  
 });  
  
 // create todo and send back all todos after creation  
 app.post('/api/todos', function(req, res) {  
 ...  
 });  
  
 // delete a todo  
 app.delete('/api/todos/:todo\_id', function(req, res) {  
 ...  
 });  
  
 // application -------------------------------------------------------------  
 app.get('\*', function(req, res) {  
 res.sendfile('./public/index.html'); // load the single view file (angular will handle the page changes on the front-end)  
 });  
  
// listen (start app with node server.js) ======================================  
app.listen(port);  
console.log("App listening on port " + port);

This single file approach is obviously not idea for expanding our app.

### Moving Forward

In the future, we want our app to-do cool things like:

* Creating more models
* Have custom configs
* Authentication using Passport for local, Facebook, and Google
* Real time updating with Socket.io
* and so much more…

In order for that to happen, we will need to move some parts around so that each section is extendable. We want flexibility to add more models, config options, features, routes, and whatever else easily.

## New File Structure

We will show off an overview of what our site structure will move to and then go through each part and explain it. **Our app will function the same as before**, it will just be better set up for the future.

<!-- new file structure -->  
  
 - app <!-- holds all our files for node components (models, routes) -->  
 ----- models  
 ---------- todo.js <!-- defines the todo model -->  
 ----- routes.js <!-- all routes will be handled here -->  
  
 - config <!-- all our configuration will be here -->  
 ----- database.js  
  
 - public <!-- holds all our files for our frontend angular application -->  
 ----- core.js <!-- all angular code for our app -->  
 ----- index.html <!-- main view -->  
  
 - package.json <!-- npm configuration to install dependencies/modules -->  
 - server.js <!-- Node configuration -->

This file structure took a lot of inspiration from [mean.io](http://mean.io/) and [Express](http://expressjs.com/guide.html). Those both have a more complex file structure and inner workings. This will be much simpler to show off the basic concepts of file separation.

### Config, Models, and Routes

Our old application had everything in server.js. You can imagine how large this one file would become. We’ll go through that file and move everything out one by one.

## Database Config config/database.js

Connecting to our database is currently handled in server.js in the **configuration** section.

// server.js (old)  
...  
  
 mongoose.connect('mongodb://node:node@mongo.onmodulus.net:27017/uwO3mypu'); // connect to mongoDB database on modulus.io  
  
...

Let’s separate out the URL for connecting to the database into our **config/database.js**.

// config/database.js  
  
 module.exports = {  
 url : 'mongodb://node:node@mongo.onmodulus.net:27017/uwO3mypu'  
 };

Now in our server.js, we can pull that database config.

// server.js (new)  
  
 // load the config  
 var database = require('./config/database');  
  
 mongoose.connect(database.url); // connect to mongoDB database on modulus.io

**Why do this?** I know it seems a little useless since it’s only the one configuration setting. It will pay off in the future though when we have multiple settings across multiple files (auth, database, application, environment).

**Understanding module.exports:** module.exports allows you to pass data from one file to another. Just using require('./config/database') doesn’t automagically give you access to those variables. module.exports exposes those variables (or functions or anything else) to other files. For a more thorough understanding, there’s this [**great article**](http://openmymind.net/2012/2/3/Node-Require-and-Exports/) by Karl Seguin.

There are many different ways to use module.exports, and we’ll be sure to write up an article in the future for all the ways to use it. Now that our config is separated out, we’ll move onto the models.

## Model app/models/todo.js

Here is our old code for the todo model in our giant server.js.

// server.js (old)  
...  
  
 // define model ================================================================  
 var Todo = mongoose.model('Todo', {  
 text : String,  
 done : Boolean  
 });  
  
...

We’ll move our current models entirely out of server.js. In our new **app/models/todo.js**, let’s use module.exports and expose our Todo mongoose model to the file that needs it whenever it is loaded using **require**.

// app/models/todo.js  
  
 // load mongoose since we need it to define a model  
 var mongoose = require('mongoose');  
  
 module.exports = mongoose.model('Todo', {  
 text : String,  
 done : Boolean  
 });

Since this model is used by our **routes** in server.js, and we are moving the routes out of server.js into app/routes.js, we won’t need to load the model there. **We will load this model in the app/routes.js file**. Just remove the lines for defining the model in server.js.

With models out of the main server.js file, our file is that much cleaner. The bulk of the code is the routes however and we’ll get that separated out next.

## Routes app/routes.js

We are going to move all of the routes in server.js out to the **app/routes.js** file. I personally prefer to hold all routes in a single file. This let’s me see a top down view of my entire application without having to dig through other files. If any of my routes start getting overly code heavy, I’ll move that code into a controller and load the controller in the routes.

Of course, this is up to your own preference to separate routes into separate files or into controllers. Some people like to separate the different routes into different files (ie api, auth, application).

Here’s our old code that we’ll move out of server.js.

// server.js (old)  
  
// this file condensed since there's so much code  
  
...  
  
// routes ======================================================================  
  
 // api ---------------------------------------------------------------------  
 // get all todos  
 app.get('/api/todos', function(req, res) {  
  
 ...  
  
 });  
  
 // create todo and send back all todos after creation  
 app.post('/api/todos', function(req, res) {  
  
 ...  
  
 });  
  
 // delete a todo  
 app.delete('/api/todos/:todo\_id', function(req, res) {  
  
 ...  
  
 });  
  
 // application -------------------------------------------------------------  
 app.get('\*', function(req, res) {  
 res.sendfile('./public/index.html'); // load the single view file (angular will handle the page changes on the front-end)  
 });  
  
...

Move all of that code out of server.js and let’ go over to our app/routes.js file.

We will **load our todo model**, and use module.exports to **expose the routes to our app**.

// app/routes.js  
  
// load the todo model  
var Todo = require('./models/todo');  
  
// expose the routes to our app with module.exports  
module.exports = function(app) {  
  
 // api ---------------------------------------------------------------------  
 // get all todos  
 app.get('/api/todos', function(req, res) {  
  
 ...  
  
 });  
  
 // create todo and send back all todos after creation  
 app.post('/api/todos', function(req, res) {  
  
 ...  
  
 });  
  
 // delete a todo  
 app.delete('/api/todos/:todo\_id', function(req, res) {  
  
 ...  
  
 });  
  
 // application -------------------------------------------------------------  
 app.get('\*', function(req, res) {  
 res.sendfile('./public/index.html'); // load the single view file (angular will handle the page changes on the front-end)  
 });  
  
};

Now that we’ve defined our routes file, we’ll just load that in our server.js and pass in our **app** variable to the function. This way the routes file has access to **app** and **express**. Now the routes we have defined are accessible in our server.js file thanks to module.exports.

// server.js  
...  
  
 // load the routes  
 require('./app/routes')(app);  
  
...

That’s it for the routes. That’s it for everything actually.

## Clean App, Clean Mind

Look at how clean our server.js is now.

// server.js (final)  
  
 // set up ======================================================================  
 var express = require('express');  
 var app = express(); // create our app w/ express  
 var mongoose = require('mongoose'); // mongoose for mongodb  
 var port = process.env.PORT || 8080; // set the port  
 var database = require('./config/database'); // load the database config  
  
 // configuration ===============================================================  
 mongoose.connect(database.url); // connect to mongoDB database on modulus.io  
  
 app.configure(function() {  
 app.use(express.static(\_\_dirname + '/public')); // set the static files location /public/img will be /img for users  
 app.use(express.logger('dev')); // log every request to the console  
 app.use(express.bodyParser()); // pull information from html in POST  
 app.use(express.methodOverride()); // simulate DELETE and PUT  
 });  
  
 // routes ======================================================================  
 require('./app/routes.js')(app);  
  
 // listen (start app with node server.js) ======================================  
 app.listen(port);  
 console.log("App listening on port " + port);

Compare that to the giant monster it used to be. That might be a bit of an exaggeration, but still, cleaner is better. Now we’ve moved **configuration**, **models**, and **routes** into their own separate files. Separation never felt so good.

Notice how everything in our public folder didn’t have to change. Our Angular side of our application, the frontend, didn’t have to change one bit. The separation of duties we established in our first app holds strong here. Node is the API level backend and Angular is the separate frontend.

If anyone has any suggestions, I’d be happy to hear them. I’m still on the long search for best practices in all things MEAN stack so any improvements are welcome.

## Moving Forward

Stay on the lookout for the next articles in this series. We’ll be going over Authentication with Passport next. We’ll be able to authenticate our users locally, with Facebook, or using Google.

This article is part of our [**Node and Angular To-Do App**](http://scotch.io/series/node-and-angular-to-do-app) series.

1. [**Creating a Single Page To-do App with Node and Angular**](http://scotch.io/tutorials/javascript/creating-a-single-page-todo-app-with-node-and-angular)
2. Node Application Organization and Structure
3. [**Angular Modules: Controllers and Services**](http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-controllers-and-services)

# **Node and Angular To-Do App: Controllers and Services 3/3**

@@http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-controllers-and-services

*·*[*angularJS*](http://scotch.io/tag/angular-js)*,* [*MEAN*](http://scotch.io/tag/mean)*,* [*node.js*](http://scotch.io/tag/node-js) [*Chris Sevilleja*](http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-controllers-and-services)  *December 9, 2013* [*25 Comments*](http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-controllers-and-services#comment-section)

In the last part of our [Node and Angular To-Do App](http://scotch.io/series/node-and-angular-to-do-app) Series, we played mostly on the Node side of things dealing with application organization and structure. Today we will be handling the Angular side of things.

Here is what we’ll be doing:

* **Organizing**: Moving our Angular $http elements into a separate *services* file.
* **Services**: Creating a service for our Todos
* **Controllers**: Moving our entire controller into its own file so its not a global controller.
* Upgrading our app to Angular 1.2.4 from 1.0.8 (just link to the new version)
* Fixing a little bug where a user could hold down the enter button keep creating the same Todo

This entire article will be on the frontend of our application. We used Node to create our backend RESTful API and that will not change. This is the beauty of creating an API. Frontend and backend remain independent of each other. This will also be great in the future if we ever want to use our API to build more than just a website. Maybe we would want to create a mobile app or something else in the future.

Let’s get started with some basic organization for our app. Nowhere near as extensive as what we did for Node. We’ll just be moving our javascript files and Angular modules into separate files.

## Organizing Our Application

Before, we had all of our Angular code in one core.js file. This isn’t what we want moving forward. We want our application to be **modular** so that our controller and all of our $http requests are in their own files.

**Why modular?** Having all of your functionality in different modules helps for many reason.

* The overall layout of your application is easier to understand.
* You can see how the parts work together since modules have to be injected before use.
* Code is reusable since all of the necessary functionality is contained inside the module.
* Testing your code is much easier

## Application Files

For this tutorial, we will only be looking in our applications public folder since that’s where all the frontend code lives.

Here is how we want our new public file structure to look.

-- public  
-------- js  
-------------- controllers  
--------------------- main.js  
-------------- services  
--------------------- todos.js  
-------------- core.js   
-------- index.html

Let’s go ahead and open up our index.html file and load up the files we need. We will also upgrade our Angular 1.0.8 to 1.2.4.

<!-- index.html -->  
...  
 <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.4/angular.min.js"></script><!-- load angular 1.2.4 -->  
  
 <script src="js/controllers/main.js"></script> <!-- load up our controller -->  
 <script src="js/services/todos.js"></script> <!-- load our todo service -->  
 <script src="js/core.js"></script> <!-- load our main application -->  
  
</head>  
...

There are many changes from 1.0.8 to 1.2.x, but they don’t seem to affect our app. If you want to see a full changelog, visit the [migration guide](http://docs.angularjs.org/guide/migration).

## To-do Service js/services/todos.js

Let’s create our service. The to-do service is meant to interact with our Node API. We want to have all the code to **get**, **create**, or **delete** a to-do inside our service. This ensures that we can test this code separate of our overall application. Let’s get all that $http code out of our main application file (core.js).

// js/services/todos.js  
angular.module('todoService', [])  
  
 // super simple service  
 // each function returns a promise object   
 .factory('Todos', function($http) {  
 return {  
 get : function() {  
 return $http.get('/api/todos');  
 },  
 create : function(todoData) {  
 return $http.post('/api/todos', todoData);  
 },  
 delete : function(id) {  
 return $http.delete('/api/todos/' + id);  
 }  
 }  
 });

That’s all there is to it. We have defined our service using .factory with three different functions. get, create and delete will all return **promise objects** that we can use in our controller.

**Declaring Services** There are many different ways to declare a service (.service, .factory and .factory). To understand the differences a little better, here’s a [Stackoverflow question and answer](http://stackoverflow.com/questions/13937318/convert-angular-http-get-function-to-a-service).

## To-do Main Controller js/controllers/main.js

Now that we have our service, let’s create our Angular module for our controller. We will be moving most of the code out of core.js into our controller file.

// js/controllers/main.js  
   
angular.module('todoController', [])  
  
 .controller('mainController', function($scope, $http) {  
 $scope.formData = {};  
  
 // when landing on the page, get all todos and show them  
 $http.get('/api/todos')  
 .success(function(data) {  
 $scope.todos = data;  
 })  
 .error(function(data) {  
 console.log('Error: ' + data);  
 });  
  
 // when submitting the add form, send the text to the node API  
 $scope.createTodo = function() {  
 $http.post('/api/todos', $scope.formData)  
 .success(function(data) {  
 $scope.formData = {}; // clear the form so our user is ready to enter another  
 $scope.todos = data;  
 })  
 .error(function(data) {  
 console.log('Error: ' + data);  
 });  
 };  
  
 // delete a todo after checking it  
 $scope.deleteTodo = function(id) {  
 $http.delete('/api/todos/' + id)  
 .success(function(data) {  
 $scope.todos = data;  
 })  
 .error(function(data) {  
 console.log('Error: ' + data);  
 });  
 };  
  
 });

We have moved our controller code out of core.js. While we now have our controller and service in their own modules, they won’t be able to work together until we inject them into our main application module.

## Getting All Modules Working Together js/core.js

To get everything working together, we just have to load our controller and services (we did that already in our index.html), and then inject our controller and service into the main module.

Make sure you move your original core.js file from the root directory into the js folder. This ensures that all our javascript code will be located in the same location.

Here is the **entire code** for our new core.js.

// js/core.js  
  
angular.module('scotchTodo', ['todoController', 'todoService']);

That’s it! You can see how easy that is to read. We have our main module scotchTodo and then we inject our controller and service.

Now that our application is set to work together again, we need to use

## Using Our Service in Our Controller

We have linked everything but we are not using our new service yet. Let’s inject that into our controller and use it!

// js/controllers/main.js  
angular.module('todoController', [])  
  
 // inject the Todo service factory into our controller  
 .controller('mainController', function($scope, $http, Todos) {  
 $scope.formData = {};  
  
 // GET =====================================================================  
 // when landing on the page, get all todos and show them  
 // use the service to get all the todos  
 Todos.get()  
 .success(function(data) {  
 $scope.todos = data;  
 });  
  
 // CREATE ==================================================================  
 // when submitting the add form, send the text to the node API  
 $scope.createTodo = function() {  
  
 // validate the formData to make sure that something is there  
 // if form is empty, nothing will happen  
 // people can't just hold enter to keep adding the same to-do anymore  
 if (!$.isEmptyObject($scope.formData)) {  
  
 // call the create function from our service (returns a promise object)  
 Todos.create($scope.formData)  
  
 // if successful creation, call our get function to get all the new todos  
 .success(function(data) {  
 $scope.formData = {}; // clear the form so our user is ready to enter another  
 $scope.todos = data; // assign our new list of todos  
 });  
 }  
 };  
  
 // DELETE ==================================================================  
 // delete a todo after checking it  
 $scope.deleteTodo = function(id) {  
 Todos.delete(id)  
 // if successful creation, call our get function to get all the new todos  
 .success(function(data) {  
 $scope.todos = data; // assign our new list of todos  
 });  
 };  
 });

As you can see, the code looks very similar to how it used to be. That is because the main thing we did was move the old$http code outside of our controller and into our service. The service will return a **promise** object so we can use the data using .success promise.

## Conclusion

Now we have organized the frontend of our application. We have separated **controller** and **service** and gotten our modules to work together via injection.

Our application should work the exact same as before, but when we eventually want to grow our application, it will be far more scalable and testable.

In the next few tutorials, we will be doing more frontend work and adding filtering and searching to our application. We will also be working on the backend again and adding Node authentication with Passport.

This article is part of our [**Node and Angular To-Do App**](http://scotch.io/series/node-and-angular-to-do-app) series.

1. [**Creating a Single Page To-do App with Node and Angular**](http://scotch.io/tutorials/javascript/creating-a-single-page-todo-app-with-node-and-angular)
2. [**Node Application Organization and Structure**](http://scotch.io/tutorials/javascript/node-and-angular-to-do-app-application-organization-and-structure)
3. Angular Modules: Controllers and Services