×

Create a Web App and RESTful API Server Using the MEAN Stack

Last updated 27 May 2020

Table of Contents

The MEAN stack is a popular web development stack made up of MongoDB, Express, Angular, and Node.js. MEAN has gained popularity because it allows developers to program in JavaScript on both the client and the server. The MEAN stack enables a perfect harmony of JavaScript Object Notation (JSON) development: MongoDB stores data in a JSON-like format, Express and Node.js facilitate easy JSON query creation, and Angular allows the client to seamlessly send and receive JSON documents.

MEAN is generally used to create browser-based web applications because Angular (client-side) and Express (server-side) are both frameworks for web apps. Another compelling use case for MEAN is the development of RESTful API servers. Creating RESTful API servers has become an increasingly important and common development task, as applications increasingly need to gracefully support a variety of end-user devices, such as mobile phones and tablets. This tutorial will demonstrate how to use the MEAN stack to rapidly create a RESTful API server.

Angular, a client-side framework, is not a necessary component for creating an API server. You could also write an Android or iOS application that runs on top of the REST API. We include Angular in this tutorial to demonstrate how it allows us to quickly create a web application that runs on top of the API server.

The application we will develop in this tutorial is a basic contact management application that supports standard CRUD (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) (Create, Read, Update, Delete) operations. First, we'll create a RESTful API server to act as an interface for querying and persisting data in a MongoDB database. Then, we'll leverage the API server to build an Angular-based web application that provides an interface for end users.

So that we can focus on illustrating the fundamental structure of a MEAN application, we will deliberately omit common functionality such as authentication, access control, and robust data validation.

Prerequisites

If you have never deployed a Node.js application to Heroku before, we recommend going through the Getting Started with Node.js on Heroku (https://devcenter.heroku.com/articles/getting-started-with-nodejs) tutorial before you begin.

Ensure that you have the following installed on your local machine:

- Heroku CLI (https://cli.heroku.com/)
- Node.js (https://nodejs.org/en/download/) version 4 or higher

This tutorial uses the Angular CLI project (https://github.com/angular/angular-cli). You can install it by running the following command:

```
$ npm install -g @angular/cli
```

The commands used in this article are for Mac/Linux. The commands are mostly the same for Windows, but you may need to substitute equivalent Windows commands in some cases.

Source code structure

The source code for this project (https://github.com/chrisckchang/mean-contactlist-angular2) is available on GitHub. Creating a new project with the Angular CLI will generate a large number of different files. We have listed the important files/folders that we'll be directly modifying below.

- `package.json`
    - A configuration file that contains the metadata for your application. When there is a `package.json` file in the root directory of the project, Heroku will use the Node.js buildpack to deploy your application.
- `app.json`

  - A manifest format for describing web apps. It declares environment variables, add-ons, and other information required to run an app on Heroku. It is required to create a “Deploy to Heroku” button.
- `server.js`
  - This file contains all the server-side code used to implement the REST API. The API is written in Node.js, using the Express framework and the MongoDB Node.js driver.
- `/src` directory
  - This folder contains all of the Angular client code for the project.

See the sample application running

To see a running version of the application this tutorial will create, you can find a running example application here: https://tranquil-shore-75468.herokuapp.com/ (https://tranquil-shore-75468.herokuapp.com/).

You can also deploy the application to Heroku with a single click:

Deploy to Heroku (https://heroku.com/deploy?template=https://github.com/chrisckchang/mean-contactlist-angular2)

Now, let’s follow the tutorial step by step.

Create a new app

Use the Angular CLI to create a new project:

```
ng new mean-contactlist-angular2
```

This command will create a directory called “mean-contactlist-angular2” which contains all the project files - this might take a while. Once the directory is created, use the `cd` into your project directory.

```
cd mean-contactlist-angular2
```

Now we’ll create an app on Heroku which prepares Heroku to receive your source code. We’ll use the Heroku CLI:

```
$ heroku create
Creating app... done, ⬢ tranquil-shore-75468
https://tranquil-shore-75468.herokuapp.com/ | https://git.heroku.com/tranquil-shore-75468.git
```

When you create an app, a git remote (called “heroku”) is also created and associated with your local git repository. Heroku also generates a random name (in this case “tranquil-shore-75468”) for your app.

Heroku recognizes an app as Node.js by the existence of a `package.json` file in the root directory. A `package.json` file was generated for you by the `ng new` command we ran earlier.

To ensure that polyfills work correctly, import the `polyfills.ts` file in `src/main.ts`. Your `src/main.ts` file should look like this:

```
import './polyfills.ts';
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));
```

Provision a MongoDB database

After you set up your application and file directory, create a MongoDB instance to persist your application's data. We'll use the mLab add-on (https://elements.heroku.com/addons/mongolab), a fully managed MongoDB service, to easily provision a new MongoDB database.

Add the mLab add-on to your app. We will use a free add-on however you will need to verify your account and enter a credit card (https://devcenter.heroku.com/articles/account-verification) to create it. To do this, at the Heroku CLI run this command to create a new Sandbox database:

```
$ heroku addons:create mongolab
```

When you create a mLab add-on, the database connection URI is stored as a config var (https://devcenter.heroku.com/articles/config-vars). Heroku config variables are equivalent to an environment variable (https://en.wikipedia.org/wiki/Environment_variable), which you can use in development and your local environment. You can access this variable in your Node.js code as `process.env.MONGODB_URI` , which we will use later in our server code.

Now that our database is ready, we can start coding.

Connect MongoDB and the app server using the Node.js driver

There are two popular MongoDB drivers that Node.js developers use: the official Node.js driver (https://docs.mongodb.com/ecosystem/drivers/node/) and an object document mapper called Mongoose (https://mongoosejs.com/) that wraps the Node.js driver (similar to a SQL ORM). Both have their advantages, but for this example we will use the official Node.js driver.

Create a new file called `server.js` .

```
touch server.js
```

In this file we'll create a new Express application and connect to our mLab database. Copy the following code into the `server.js` file.

```
var express = require("express");
var bodyParser = require("body-parser");
var mongodb = require("mongodb");
var ObjectID = mongodb.ObjectID;

var CONTACTS_COLLECTION = "contacts";

var app = express();
app.use(bodyParser.json());

// Create a database variable outside of the database connection callback to reuse the connection pool i
var db;

// Connect to the database before starting the application server.
mongodb.MongoClient.connect(process.env.MONGODB_URI || "mongodb://localhost:27017/test", function (err,
  if (err) {
    console.log(err);
    process.exit(1);
  }

  // Save database object from the callback for reuse.
  db = client.db();
  console.log("Database connection ready");

  // Initialize the app.
  var server = app.listen(process.env.PORT || 8080, function () {
    var port = server.address().port;
    console.log("App now running on port", port);
  });
});

// CONTACTS API ROUTES BELOW
```

There are a few things to note regarding connecting to the database:

- We want to use our database connection pool (http://blog.mlab.com/2013/11/deep-dive-into-connection-pooling/) as often as possible to best manage our available resources. We initialize the `db` variable in the global scope so that the connection can be used by all the route handlers.
- We initialize the app only after the database connection is ready. This ensures that the application won’t crash or error out by trying database operations before the connection is established.

Note that our Express app requires a few different libraries. We’ll want to install these libraries and save the dependencies to our `package.json` file so that they will also be installed when we deploy the application to Heroku.

```
npm install mongodb express body-parser --save
```

Now our app and database are connected. Next we will implement the RESTful API server by defining all the endpoints.

Create a RESTful API server with Node.js and Express

As our first step in creating the API, we define the endpoints (or data) we want to expose. Our contact list app will allow users to perform CRUD operations on their contacts.

The endpoints we’ll need are:

`/api/contacts`

Method	Description
GET	Find all contacts
POST	Create a new contact

`/api/contacts/:id`

Method	Description
GET	Find a single contact by ID
PUT	Update entire contact document
DELETE	Delete a contact by ID

Now we'll add the routes to our `server.js` file:

```
// CONTACTS API ROUTES BELOW

// Generic error handler used by all endpoints.
function handleError(res, reason, message, code) {
  console.log("ERROR: " + reason);
  res.status(code || 500).json({"error": message});
}

/*  "/api/contacts"
 *    GET: finds all contacts
 *    POST: creates a new contact
 */

app.get("/api/contacts", function(req, res) {
});

app.post("/api/contacts", function(req, res) {
});

/*  "/api/contacts/:id"
 *    GET: find contact by id
 *    PUT: update contact by id
 *    DELETE: deletes contact by id
 */

app.get("/api/contacts/:id", function(req, res) {
});

app.put("/api/contacts/:id", function(req, res) {
});

app.delete("/api/contacts/:id", function(req, res) {
});
```

The code creates a skeleton for all of the API endpoints defined above.

Implement the API endpoints

Next, we'll add in database logic to properly implement these endpoints.

For testing purposes, we'll first implement the GET and POST endpoints for "/api/contacts". This will allow us to test getting contacts from the database and saving new contacts to the database. Each contact will have the following schema. We will use this format later when defining the `Contact` class.

```
{
  "_id": <ObjectId>,
  "name": <string>,
  "email": <string>,
  "phone": {
    "mobile": <string>,
    "work": <string>
  }
}
```

The following code implements the /contacts POST request:

```
/*  "/api/contacts"
 *    GET: finds all contacts
 *    POST: creates a new contact
 */

app.get("/api/contacts", function(req, res) {
  db.collection(CONTACTS_COLLECTION).find({}).toArray(function(err, docs) {
    if (err) {
      handleError(res, err.message, "Failed to get contacts.");
    } else {
      res.status(200).json(docs);
    }
  });
});

app.post("/api/contacts", function(req, res) {
  var newContact = req.body;
  newContact.createDate = new Date();

  if (!req.body.name) {
    handleError(res, "Invalid user input", "Must provide a name.", 400);
  } else {
    db.collection(CONTACTS_COLLECTION).insertOne(newContact, function(err, doc) {
      if (err) {
        handleError(res, err.message, "Failed to create new contact.");
      } else {
        res.status(201).json(doc.ops[0]);
      }
    });
  }
});
```

To test the endpoints, we'll deploy the code to Heroku. Navigate back to the root project directory and change the "start" script in your `package.json` .

```
"start": "node server.js"
```

This will instruct Heroku to run the `server.js` file to start the application. Then we'll push the application to Heroku.

```
$ git add .
$ git commit -m 'test API endpoints'
$ git push heroku master
```

The application is now live. We’ll use cURL (https://curl.haxx.se/) to issue a POST request:

```
$ curl -H "Content-Type: application/json" -d '{"name":"mLab Support", "email": "support@mlab.com"}' h
ttp://your-app-name.herokuapp.com/api/contacts
```

We haven’t created our web app yet, but you can confirm that the data was successfully saved to the database by copying the above URL in your browser: `http://your-app-name.herokuapp.com/api/contacts` . Your new contact should be displayed in your browser window.

Here is the final version of the `server.js` file (for now), which implements all of the endpoints. Copy this into your `server.js` file before moving on to the next step.

```
var express = require("express");
var bodyParser = require("body-parser");
var mongodb = require("mongodb");
var ObjectID = mongodb.ObjectID;

var CONTACTS_COLLECTION = "contacts";

var app = express();
app.use(bodyParser.json());

// Create a database variable outside of the database connection callback to reuse the connection pool i
var db;

// Connect to the database before starting the application server.
mongodb.MongoClient.connect(process.env.MONGODB_URI || "mongodb://localhost:27017/test", function (err,
  if (err) {
    console.log(err);
    process.exit(1);
  }

  // Save database object from the callback for reuse.
  db = client.db();
  console.log("Database connection ready");

  // Initialize the app.
  var server = app.listen(process.env.PORT || 8080, function () {
    var port = server.address().port;
    console.log("App now running on port", port);
  });
});

// CONTACTS API ROUTES BELOW

// Generic error handler used by all endpoints.
function handleError(res, reason, message, code) {
  console.log("ERROR: " + reason);
  res.status(code || 500).json({"error": message});
}

/*  "/api/contacts"
 *    GET: finds all contacts
 *    POST: creates a new contact
 */

app.get("/api/contacts", function(req, res) {
  db.collection(CONTACTS_COLLECTION).find({}).toArray(function(err, docs) {
    if (err) {
      handleError(res, err.message, "Failed to get contacts.");
    } else {
      res.status(200).json(docs);
    }
  });
});

app.post("/api/contacts", function(req, res) {
  var newContact = req.body;
  newContact.createDate = new Date();

  if (!req.body.name) {
    handleError(res, "Invalid user input", "Must provide a name.", 400);
  } else {
    db.collection(CONTACTS_COLLECTION).insertOne(newContact, function(err, doc) {
      if (err) {
        handleError(res, err.message, "Failed to create new contact.");
      } else {
        res.status(201).json(doc.ops[0]);
      }
    });
  }
});

/*  "/api/contacts/:id"
 *    GET: find contact by id
 *    PUT: update contact by id
 *    DELETE: deletes contact by id
 */

app.get("/api/contacts/:id", function(req, res) {
  db.collection(CONTACTS_COLLECTION).findOne({ _id: new ObjectID(req.params.id) }, function(err, doc) {
    if (err) {
      handleError(res, err.message, "Failed to get contact");
    } else {
      res.status(200).json(doc);
    }
  });
});

app.put("/api/contacts/:id", function(req, res) {
  var updateDoc = req.body;
  delete updateDoc._id;

  db.collection(CONTACTS_COLLECTION).updateOne({_id: new ObjectID(req.params.id)}, updateDoc, function(e
    if (err) {
      handleError(res, err.message, "Failed to update contact");
    } else {
      updateDoc._id = req.params.id;
      res.status(200).json(updateDoc);
    }
  });
});

app.delete("/api/contacts/:id", function(req, res) {
  db.collection(CONTACTS_COLLECTION).deleteOne({_id: new ObjectID(req.params.id)}, function(err, result)
    if (err) {
      handleError(res, err.message, "Failed to delete contact");
    } else {
      res.status(200).json(req.params.id);
    }
  });
});
```

Set up the Angular project structure

Now that our RESTful server is complete, we can set up the structure for our Angular web application. The `src/app` folder holds the Angular project code, so we'll put our work in there.

Create a subdirectory called `src/app/contacts` . The `contacts` folder will contain the application logic for displaying and handling contacts.

```
mkdir src/app/contacts
```

Next we'll create a contact class file that will help us keep our schema consistent with what we defined previously in the Implement the API endpoints (https://devcenter.heroku.com/articles/mean-apps-restful-api#implement-the-api-endpoints) step.

```
ng generate class contacts/contact
```

The contact class will be used by our components. Each component controls a template and is where we define our application logic.

```
ng generate component contacts/contact-details
ng generate component contacts/contact-list
```

Finally, we'll create an Angular service that will be used by our components to send and receive data.

```
ng generate service contacts/contact
```

When you're finished, the project structure should mirror the reference project (https://github.com/chrisckchang/mean-contactlist-angular2/tree/master/src/app/contacts) folder and file structure.

Define the contact class

Navigate to `src/app/contacts/contact.ts` and insert the following code:

```
export class Contact {
  _id?: string;
  name: string;
  email: string;
  phone: {
    mobile: string;
    work: string;
  }
}
```

MongoDB by default creates an `_id` ObjectId (https://docs.mongodb.com/manual/reference/method/ObjectId/) field for each document that is inserted into the database. When we create a contact in our client-side Angular app we'll leave the `_id` field blank because it will be auto-generated on the server side.

Create the contact service to make requests to the API server

Our service will act as the client-side wrapper for the RESTful API endpoints that the web application needs. Change your `src/app/contacts/contact.service.ts` to the following:

```
import { Injectable } from '@angular/core';
import { Contact } from './contact';
import { Http, Response } from '@angular/http';

@Injectable()
export class ContactService {
    private contactsUrl = '/api/contacts';

    constructor (private http: Http) {}

    // get("/api/contacts")
    getContacts(): Promise<void | Contact[]> {
      return this.http.get(this.contactsUrl)
                 .toPromise()
                 .then(response => response.json() as Contact[])
                 .catch(this.handleError);
    }

    // post("/api/contacts")
    createContact(newContact: Contact): Promise<void | Contact> {
      return this.http.post(this.contactsUrl, newContact)
                 .toPromise()
                 .then(response => response.json() as Contact)
                 .catch(this.handleError);
    }

    // get("/api/contacts/:id") endpoint not used by Angular app

    // delete("/api/contacts/:id")
    deleteContact(delContactId: String): Promise<void | String> {
      return this.http.delete(this.contactsUrl + '/' + delContactId)
                 .toPromise()
                 .then(response => response.json() as String)
                 .catch(this.handleError);
    }

    // put("/api/contacts/:id")
    updateContact(putContact: Contact): Promise<void | Contact> {
      var putUrl = this.contactsUrl + '/' + putContact._id;
      return this.http.put(putUrl, putContact)
                 .toPromise()
                 .then(response => response.json() as Contact)
                 .catch(this.handleError);
    }

    private handleError (error: any) {
      let errMsg = (error.message) ? error.message :
      error.status ? `${error.status} - ${error.statusText}` : 'Server error';
      console.error(errMsg); // log to console instead
    }
}
```

At the top of the `contact.service.ts` file we import the contact class that we created along with the built-in Angular $http service. By default, $http requests return an Angular Observable.

Note that with the $http service we use relative URL paths (e.g., "/api/contacts") as opposed to absolute paths like "app-name.herokuapp.com/api/contacts".

Create the contact list template and component

To display a contact list to the user, we'll need a template (or view) and the application logic to control that template. Let's first create the template by modifying `src/app/contacts/contact-list/contact-list.component.html`.

```
<div class="row">
  <div class="col-md-5">
    <h2>Contacts</h2>
    <ul class="list-group">
      <li class="list-group-item"
        *ngFor="let contact of contacts"
        (click)="selectContact(contact)"
        [class.active]="contact === selectedContact">
        {{contact.name}}
      </li>
    </ul>
    <button class="btn btn-warning" (click)="createNewContact()">New</button>
  </div>
  <div class="col-md-5 col-md-offset-2">
    <contact-details
      [contact]="selectedContact"
      [createHandler]="addContact"
      [updateHandler]="updateContact"
      [deleteHandler]="deleteContact">
    </contact-details>
  </div>
</div>
```

This template displays a contact list and also includes the contact-details template, which we'll implement in the next step.

Next, we'll add in our application logic to the `contact-list.component.ts` file.

```
import { Component, OnInit } from '@angular/core';
import { Contact } from '../contact';
import { ContactService } from '../contact.service';
import { ContactDetailsComponent } from '../contact-details/contact-details.component';

@Component({
  selector: 'contact-list',
  templateUrl: './contact-list.component.html',
  styleUrls: ['./contact-list.component.css'],
  providers: [ContactService]
})

export class ContactListComponent implements OnInit {

  contacts: Contact[]
  selectedContact: Contact

  constructor(private contactService: ContactService) { }

  ngOnInit() {
     this.contactService
      .getContacts()
      .then((contacts: Contact[]) => {
        this.contacts = contacts.map((contact) => {
          if (!contact.phone) {
            contact.phone = {
              mobile: '',
              work: ''
            }
          }
          return contact;
        });
      });
  }

  private getIndexOfContact = (contactId: String) => {
    return this.contacts.findIndex((contact) => {
      return contact._id === contactId;
    });
  }

  selectContact(contact: Contact) {
    this.selectedContact = contact
  }

  createNewContact() {
    var contact: Contact = {
      name: '',
      email: '',
      phone: {
        work: '',
        mobile: ''
      }
    };

    // By default, a newly-created contact will have the selected state.
    this.selectContact(contact);
  }

  deleteContact = (contactId: String) => {
    var idx = this.getIndexOfContact(contactId);
    if (idx !== -1) {
      this.contacts.splice(idx, 1);
      this.selectContact(null);
    }
    return this.contacts;
  }

  addContact = (contact: Contact) => {
    this.contacts.push(contact);
    this.selectContact(contact);
    return this.contacts;
  }

  updateContact = (contact: Contact) => {
    var idx = this.getIndexOfContact(contact._id);
    if (idx !== -1) {
      this.contacts[idx] = contact;
      this.selectContact(contact);
    }
    return this.contacts;
  }
}
```

When the application is initialized, `ngOnInit()` is called. Upon app start, we use contact service to retrieve the full contact list from the API server. Once the contact list is retrieved, it is stored into a local copy of the contact list. It's important to store a local copy of the contact list so that we can dynamically change the contact list whenever a new user is created, modified, or deleted without having to make extra HTTP requests to the API server.

Create the contact details template and component

The contact details template allows users to create, view, modify, and delete contacts from the contact list. Whenever a change to a contact is made, we need to send the update to the server but also update our local contact list. Taking a look back at our `contact-list.component.html` code, you'll notice that we pass in some inputs to the contact-details template.

```
<div class="col-md-5 col-md-offset-2">
  <contact-details
    [contact]="selectedContact"
    [createHandler]="addContact"
    [updateHandler]="updateContact"
    [deleteHandler]="deleteContact">
  </contact-details>
</div>
```

The `[contact]` input corresponds to the particular contact that the user clicks on in the UI. The three handler functions are necessary to allow the `contact-details` component to modify the local copy of the contact list created by the `contact-list` component.

Now we'll create the `contact-details.component.html` template.

```
<div *ngIf="contact" class="row">
  <div class="col-md-12">
    <h2 *ngIf="contact._id">Contact Details</h2>
    <h2 *ngIf="!contact._id">New Contact</h2>
  </div>
</div>
<div *ngIf="contact" class="row">
  <form class="col-md-12">
    <div class="form-group">
      <label for="contact-name">Name</label>
      <input class="form-control" name="contact-name" [(ngModel)]="contact.name" placeholder="Name"/>
    </div>
    <div class="form-group">
      <label for="contact-email">Email</label>
      <input class="form-control" name="contact-email" [(ngModel)]="contact.email" placeholder="support@
    </div>
    <div class="form-group">
      <label for="contact-phone-mobile">Mobile</label>
      <input class="form-control" name="contact-phone-mobile" [(ngModel)]="contact.phone.mobile" placeho
    </div>
    <div class="form-group">
      <label for="contact-phone-work">Work</label>
      <input class="form-control" name="contact-phone-work" [(ngModel)]="contact.phone.work" placeholder
    </div>
    <button class="btn btn-primary" *ngIf="!contact._id" (click)="createContact(contact)">Create</buttor
    <button class="btn btn-info" *ngIf="contact._id" (click)="updateContact(contact)">Update</button>
    <button class="btn btn-danger" *ngIf="contact._id" (click)="deleteContact(contact._id)">Delete</butt
  </form>
</div>
```

Note that our template calls three functions: `createContact()` , `updateContact()` , and `deleteContact()` . We'll need to implement these functions in our component file `contact-details.component.ts` . Let's change our component file to the following.

```
import { Component, Input } from '@angular/core';
import { Contact } from '../contact';
import { ContactService } from '../contact.service';

@Component({
  selector: 'contact-details',
  templateUrl: './contact-details.component.html',
  styleUrls: ['./contact-details.component.css']
})

export class ContactDetailsComponent {
  @Input()
  contact: Contact;

  @Input()
  createHandler: Function;
  @Input()
  updateHandler: Function;
  @Input()
  deleteHandler: Function;

  constructor (private contactService: ContactService) {}

  createContact(contact: Contact) {
    this.contactService.createContact(contact).then((newContact: Contact) => {
      this.createHandler(newContact);
    });
  }

  updateContact(contact: Contact): void {
    this.contactService.updateContact(contact).then((updatedContact: Contact) => {
      this.updateHandler(updatedContact);
    });
  }

  deleteContact(contactId: String): void {
    this.contactService.deleteContact(contactId).then((deletedContactId: String) => {
      this.deleteHandler(deletedContactId);
    });
  }
}
```

Update the main app template to display the contact list

With our contact list and contact details components created, we now need to configure our app to display these templates to the user. The default template is `app.component.html` , which we'll change to the following.

```
<div class="container">
  <contact-list></contact-list>
</div>
```

To add some style to our app, we'll add bootstrap to our project. Add the following line inside the head tag of `src/index.html` .

```
<!-- Latest compiled and minified CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" inte
```

Finalize deployment configuration and deploy to Heroku

With our Angular code finished, we're now ready to deploy the application. We'll need to install the Angular CLI so that our remote Heroku deployment can use it:

```
npm install --save @angular/cli @angular/compiler-cli
```

The finalized `package.json` file should look like the following.

```
{
  "name": "mean-contactlist-angular2",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "node server.js",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e",
    "postinstall": "ng build --output-path dist"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "^6.0.3",
    "@angular/cli": "^6.0.8",
    "@angular/common": "^6.0.3",
    "@angular/compiler": "^6.0.3",
    "@angular/compiler-cli": "^6.0.7",
    "@angular/core": "^6.0.3",
    "@angular/forms": "^6.0.3",
    "@angular/http": "^6.0.3",
    "@angular/platform-browser": "^6.0.3",
    "@angular/platform-browser-dynamic": "^6.0.3",
    "@angular/router": "^6.0.3",
    "body-parser": "^1.18.3",
    "core-js": "^2.5.4",
    "express": "^4.16.3",
    "mongodb": "^3.1.1",
    "rxjs": "^6.0.0",
    "zone.js": "^0.8.26"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~0.6.8",
    "@angular/language-service": "^6.0.3",
    "@types/jasmine": "~2.8.6",
    "@types/jasminewd2": "~2.0.3",
    "@types/node": "~8.9.4",
    "codelyzer": "~4.2.1",
    "jasmine-core": "~2.99.1",
    "jasmine-spec-reporter": "~4.2.1",
    "karma": "~1.7.1",
    "karma-chrome-launcher": "~2.2.0",
    "karma-coverage-istanbul-reporter": "~2.0.0",
    "karma-jasmine": "~1.1.1",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~5.3.0",
    "ts-node": "~5.0.1",
    "tslint": "~5.9.1",
    "typescript": "~2.7.2"
  }
}
```

There are a few changes to note. We:

- added `ng build --output-path dist` as a `"postinstall"` script. This will build the Angular application after library dependencies have been installed.
- changed `"start"` script from `ng serve` to `node server.js`. The `ng serve` command generates and serves the Angular application. However, our project also consists of the Express API server that we need to run.
- moved `@angular/cli` and `@angular/compiler-cli` from `"devDependencies"` to `"dependencies"`.

The `ng build` command stores the Angular build artifacts in the `dist/` directory. We'll configure our Express application to serve the Angular app by creating a link to the `dist/` directory. Modify the `server.js` code to include the last two lines of code:

```
var express = require("express");
var bodyParser = require("body-parser");
var mongodb = require("mongodb");
var ObjectID = mongodb.ObjectID;

var CONTACTS_COLLECTION = "contacts";

var app = express();
app.use(bodyParser.json());

// Create link to Angular build directory
var distDir = __dirname + "/dist/";
app.use(express.static(distDir));

// Rest of server.js code below
```

Finally, we'll modify our `app.module.ts` file to import the HttpModule and FormsModule:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';
import { ContactDetailsComponent } from './contacts/contact-details/contact-details.component';
import { ContactListComponent } from './contacts/contact-list/contact-list.component';

@NgModule({
  declarations: [
    AppComponent,
    ContactDetailsComponent,
    ContactListComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Complete the project and deploy

```
$ git add .
$ git commit -m 'implement all API endpoints and create angular web app'
$ git push heroku master
```

Now that the web application component is complete, you can view your app by opening the website from the CLI:

```
$ heroku open
```

Summary

In this tutorial, you learned how to:

- create a RESTful API server in Express and Node.js.
- connect a MongoDB database to the API server for querying and persisting data.
- create a rich web app using Angular.

We hope that you have seen the power of the MEAN stack to enable the development of common components for today's web applications.

Notes on scaling

If you are running a production MEAN application on Heroku, you will need to scale both your application and database as your traffic increases and data size grows. Refer to the Optimizing Node.js Application Concurrency (https://devcenter.heroku.com/articles/node-concurrency) article for best practices on scaling your application. To upgrade your database, see the mLab add-on documentation (https://devcenter.heroku.com/articles/mongolab).

Optional next steps

This app intentionally omits details you would want to include in a real production application. In particular, we do not implement a user model, user authentication, or input validation. Consider adding these features as an additional exercise. If you have any questions about this tutorial you can reach the mLab team at support@mlab.com (mailto:support@mlab.com).