

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

CONTACT LIST APPLICATION
SUNNY PATEL

Table of Contents

Introduction	2
GitHub Link for the project.....	2
Prerequisites	2
Source code structure	2
Sample Application Running	3
Connect the MongoDB on the app server using the Node.js driver	3
Create a RESTful API server with Node.js and Express	5
Implement the API Endpoints.....	6
Create a new app.....	13
Setup the Angular Project Structure	13
Define the Contact Class	14
Create the contact service to make requests to the API server	14
Create a contact list template and component	15
Create a contact details template and component.....	18
Update the main app template to display the contact list.....	21
Finalize the deployment configuration and run the app	23
Final Project Structure	28
Complete the project and run or deploy	29
Summary.....	29
References:.....	30

Introduction

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](#) (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.

GitHub Link for the project

<https://github.com/sunny3p/ContactApp.git>

Prerequisites

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

- [Node.js](#) version 4 or higher
- [Angular CLI project](#): You can install it by running the following command:

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

Source code structure

The [source code for this project](#) 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, We will use the Node.js buildpack to deploy our application.
- `app.json`
 - A manifest format for describing web apps. It declares environment variables, addons, and other information required to run an app on local machine.
- `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.

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/>.

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

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

There are two popular MongoDB drivers that Node.js developers use: the official [Node.js driver](#) and an object document mapper called [Mongoose](#) that wraps the Node.js driver (similar to a SQL ORM). Both have their advantages, but for this example we will use the Mongoose.

Create a new file called `server.js`.

For Mac:

```
touch server.js
```

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

```
let express = require("express");
```

```
let bodyParser = require("body-parser");
let mongoose = require("mongoose");

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

// MongoDB connection url
let url = process.env.MONGODB_URI || "mongodb://localhost:27017/contactlist";

let option = {
  useUnifiedTopology: true,
  useNewUrlParser: true
};

// Connect to the database before starting the application server.
mongoose.connect(url, option, function (err) {
  if (err) {
    process.exit(1);
    return console.log('Mongoose - connection error:', err);
  }
  console.log('Connect Successfully');
  // Initialize the app.
  let server = app.listen(process.env.PORT || 8080, function () {
    let port = server.address().port;
    console.log("App now running on port", port);
  });
});
```

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

- We want to use our database connection pool as often as possible to best manage our available resources.
- 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

```
npm init -y
npm install express body-parser --save
npm install mongoose --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 a contact by ID
DELETE	Delete a contact by ID

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

```
// CONTACTS API ENDPOINTS BELOW
/* "/api/contacts"
 *   GET: finds all contacts
 *   POST: creates a new contact
 */

app.get("/api/contacts", contacts.getAllContacts);

app.post("/api/contacts", contacts.addContacts);

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

```
*    DELETE: deletes contact by id
*/
app.get("/api/contacts/:id", contacts.getContactID);

app.put("/api/contacts/:id", contacts.updateContactID);

app.delete("/api/contacts/:id", contacts.deleteContactID);
```

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` schema.

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

The following steps to be followed for creating contact schema:

- i. Create a models folder
- ii. Inside models folder create a file name `Contacts.js`

The following code implements the Contact Schema:

```
const mongoose = require('mongoose');
const contactSchema = new mongoose.Schema({
  _id: mongoose.Schema.Types.ObjectId,
  name: {
    type: String,
    required: true
  },
  phone: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true
  }
});
```

```
email: String,  
phone: {  
  mobile: String,  
  work: String  
}  
});  
module.exports = mongoose.model('Contact', contactSchema);
```

Now let's create all routes handler. To create it please follow the below steps:

- i. Create a routers folder
- ii. Inside routers folder create a file name contacts.js

The following code implements the /contacts GET and POST request:

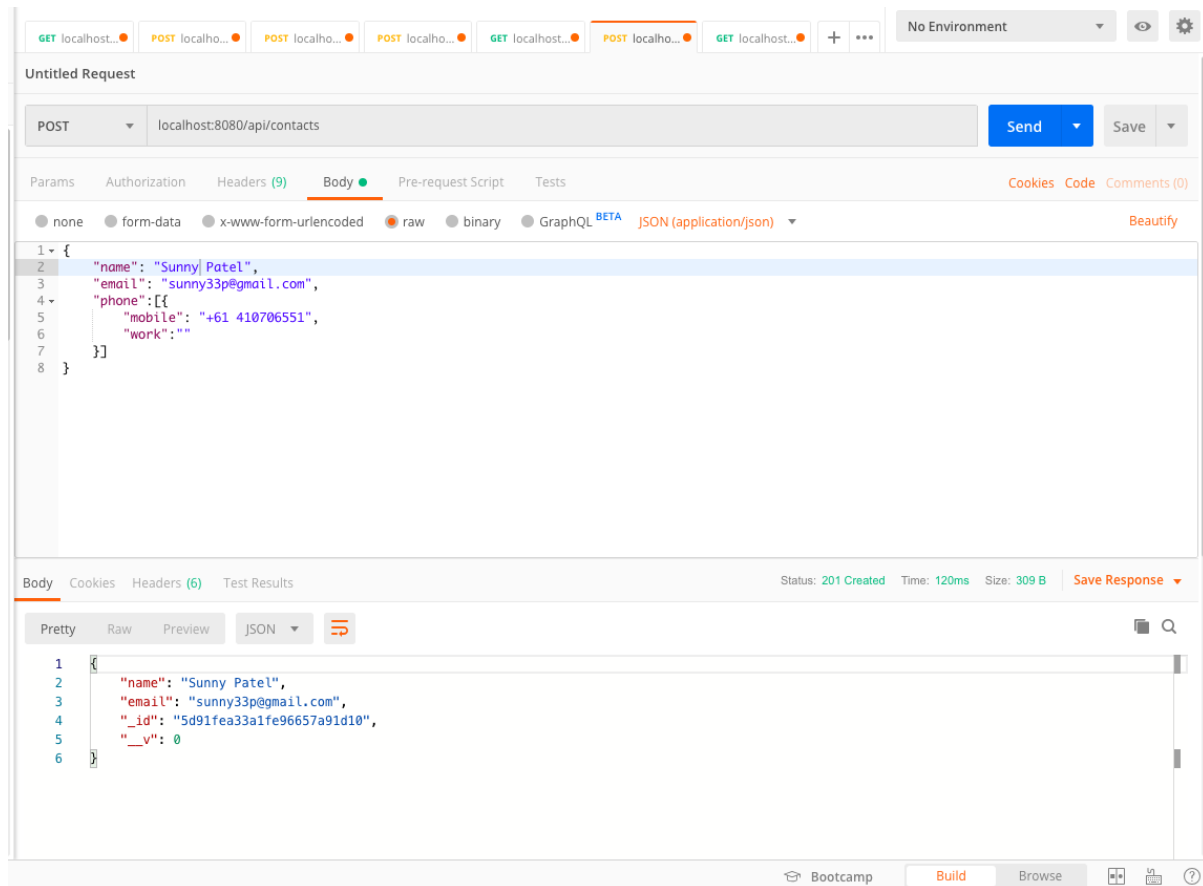
```
const mongoose = require("mongoose");  
const Contacts = require("../models/Contacts");  
  
// Generic error handler used by all endpoints.  
function handleError(res, reason, message, code) {  
  console.log("ERROR: " + reason);  
  res.status(code || 500).json({  
    "error": message  
  });  
}  
  
module.exports = {  
  getAllContacts: function (req, res) {  
    Contacts.find({})  
      .exec(function (err, contacts) {  
        if (err) {  
          handleError(res, err.message, "Failed to get contacts.");  
        } else {  
          res.status(200).json(contacts);  
        }  
      });  
  },  
  addContacts: function (req, res) {  
    let newContactDetails = req.body;
```



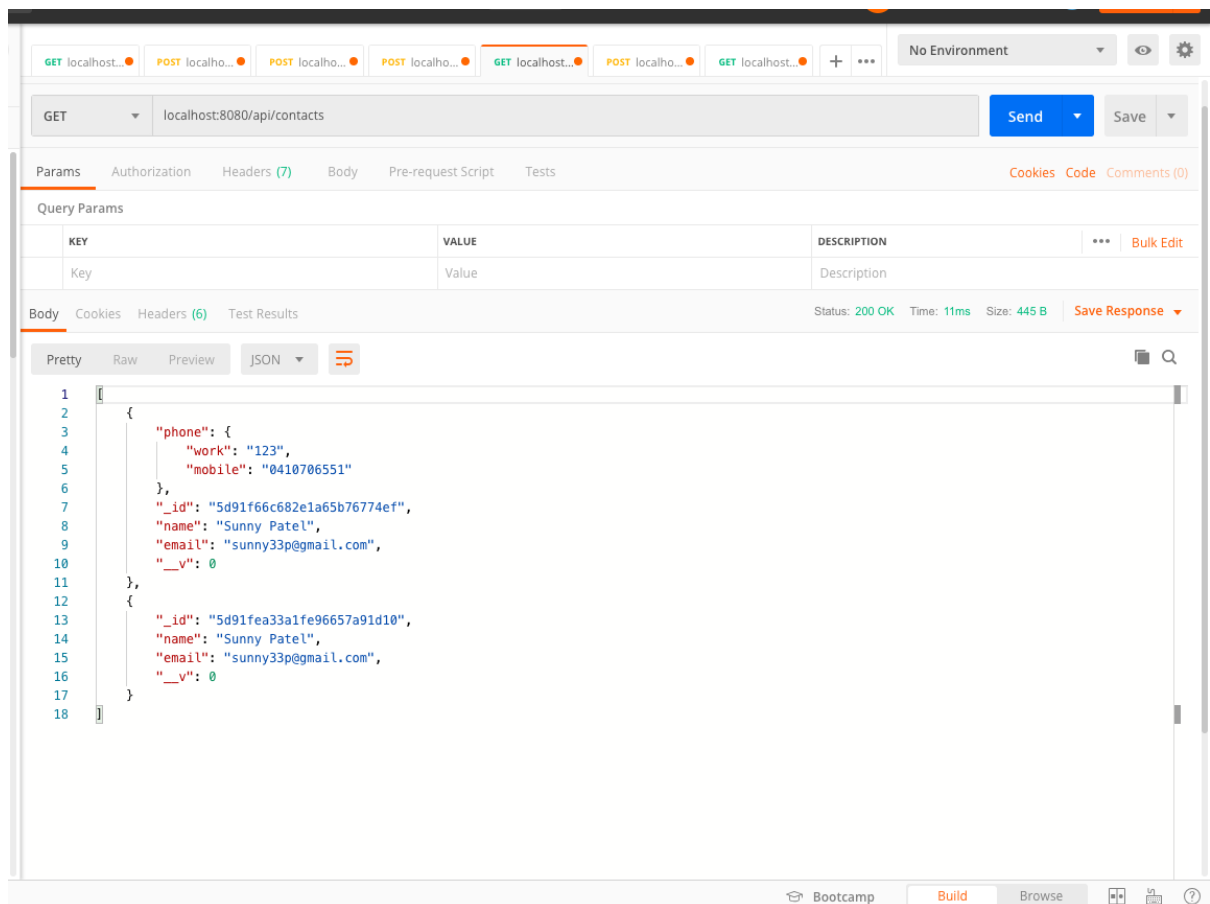
```
newContactDetails._id = new mongoose.Types.ObjectId();
if (!newContactDetails.name) {
  handleError(res, "Invalid user input", "Must provide a name.", 400);
} else {
  let contact = new Contacts(newContactDetails);
  contact.save(function (err) {
    if (err) {
      handleError(res, err.message, "Failed to create new contact.");
    } else {
      res.status(201).json(contact);
    }
  });
}
}
```

To test the endpoints, we'll run the code using node server.js. After that open postman to test our GET and POST request.

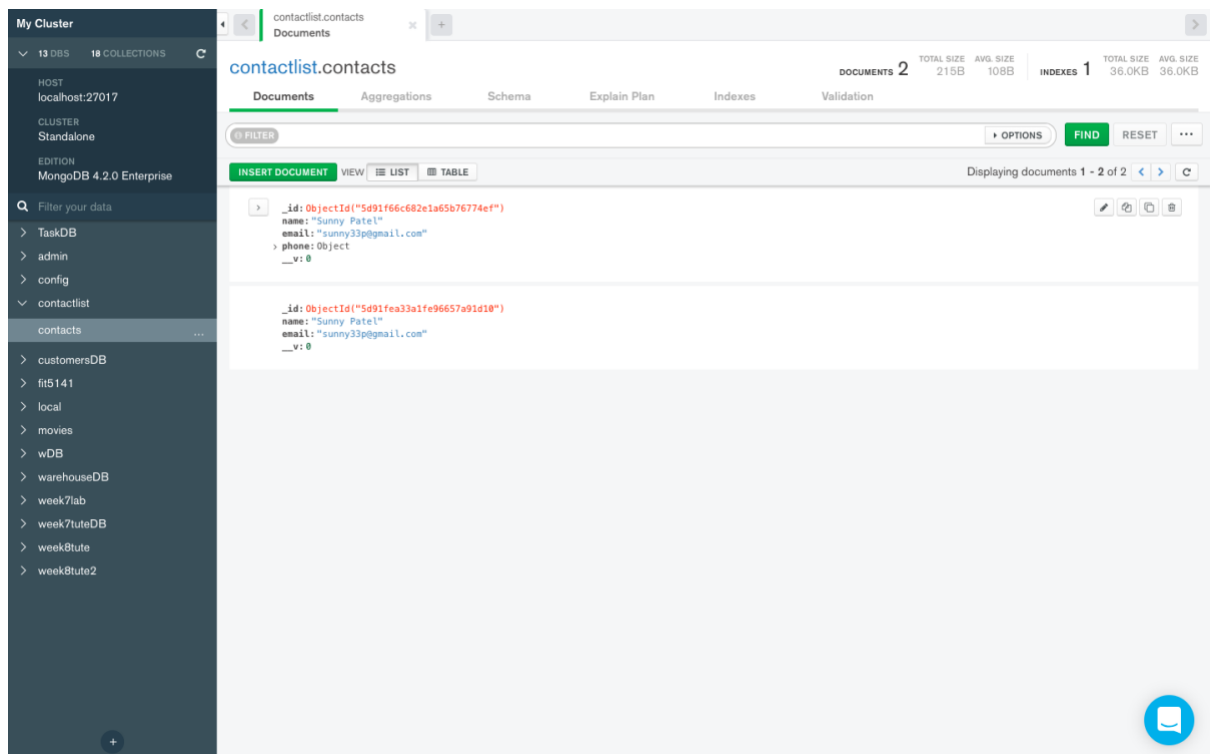
- Testing POST endpoint



- Testing GET endpoint



○ Our MongoDB Compass



Here is the final version of the `contacts.js` file, which implements all of the endpoints.

Copy this into your `contacts.js` file before moving on to the next step.

```
const mongoose = require("mongoose");
const Contacts = require("../models/Contacts");

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

module.exports = {
  getAllContacts: function (req, res) {
    Contacts.find({})
      .exec(function (err, contacts) {
        if (err) {
          handleError(res, err.message, "Failed to get contacts.");
        } else {
          res.status(200).json(contacts);
        }
      });
  },
  addContacts: function (req, res) {
    let newContactDetails = req.body;
    newContactDetails._id = new mongoose.Types.ObjectId();
    if (!newContactDetails.name) {
      handleError(res, "Invalid user input", "Must provide a name.", 400);
    } else {
      let contact = new Contacts(newContactDetails);
      contact.save(function (err) {
        if (err) {
          handleError(res, err.message, "Failed to create new contact.");
        } else {
          res.status(201).json(contact);
        }
      });
    }
  }
};
```

```
    }
  });
}
},
getContactID: function (req, res) {
  console.log(req.params);
  Contacts.findOne({
    _id: req.params.id
  }, function (err, contact) {
    if (err) {
      handleError(res, err.message, "Failed to get contact");
    } else {
      res.status(200).json(contact);
    }
  });
},
updateContactID: function (req, res) {
  let updateDoc = req.body;
  delete updateDoc._id;
  Contacts.updateOne({
    _id: req.params.id
  }, updateDoc,
  function (err, contact) {
    if (err) {
      handleError(res, err.message, "Failed to update contact");
    } else {
      updateDoc._id = req.params.id;
      res.status(200).json(updateDoc);
    }
  });
},
deleteContactID: function (req, res) {
  Contacts.deleteOne({
    _id: req.params.id
  }, function (err, result) {
    if (err) {
      handleError(res, err.message, "Failed to delete contact");
    } else {
```

```
        res.status(200).json(req.params.id);
    }
    });
}
};
```

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
```

Setup 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 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](#) 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](#) 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 { HttpClient, HttpHeaders } from "@angular/common/http";  
const httpOptions = {  
  headers: new HttpHeaders({ "Content-Type": "application/json" })  
};  
@Injectable({  
  providedIn: "root"
```

```
})  
export class ContactService {  
  constructor(private http: HttpClient) {}  
  result: any;  
  getAllContacts() {  
    return this.http.get("/api/contacts");  
  }  
  getContact(id: string) {  
    let url: string = "/api/contacts/" + id;  
    return this.http.get(url);  
  }  
  createContact(data) {  
    return this.http.post("/api/contacts", data, httpOptions);  
  }  
  updateContact(id, data) {  
    let url: string = "/api/contacts/" + id;  
    return this.http.put(url, data, httpOptions);  
  }  
  deleteContact(id) {  
    let url = "/api/contacts/" + id;  
    return this.http.delete(url, httpOptions);  
  }  
}
```

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"`)

Create a 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>
```



```

<button class="btn btn-warning" (click)="createNewContact()">New</button>
  <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>
</div>
<div class="col-md-5 col-md-offset-2">
  <app-contact-details
    [contact]="selectedContact"
    [createHandler]="addContact"
    [updateHandler]="updateContact"
    [deleteHandler]="deleteContact">
  </app-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 { ContactService } from "../../contact.service";
import { Component, OnInit } from "@angular/core";
import { Contact } from "../contact";

@Component({
  selector: "app-contact-list",
  templateUrl: "./contact-list.component.html",
  styleUrls: ["./contact-list.component.css"]
})
export class ContactListComponent implements OnInit {
  contacts: Contact[];
  selectedContact: Contact;

  constructor(private contactService: ContactService) {}

```

```
ngOnInit() {
  this.contactService.getAllContacts().subscribe((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) => {
    let 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 a 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">
  <app-contact-details
    [contact]="selectedContact"
    [createHandler]="addContact"
    [updateHandler]="updateContact"
    [deleteHandler]="deleteContact">
  </app-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@mlab.com"/>
    </div>
    <div class="form-group">
      <label for="contact-phone-mobile">Mobile</label>
      <input class="form-control" name="contact-phone-mobile"
[(ngModel)]="contact.phone.mobile" placeholder="1234567890"/>
    </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="0123456789"/>
  </div>
  <button class="btn btn-primary" *ngIf="!contact._id"
  (click)="createContact(contact)">Create</button>
  <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</button>
</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 { ContactService } from "../../contact.service";
import { Component, OnInit, Input } from "@angular/core";
import { Contact } from "../contact";

@Component({
  selector: "app-contact-details",
  templateUrl: "./contact-details.component.html",
  styleUrls: ["./contact-details.component.css"]
})
export class ContactDetailsComponent implements OnInit {
  @Input()
  contact: Contact;

  @Input()
  createHandler: Function;

  @Input()
  updateHandler: Function;

  @Input()
  deleteHandler: Function;

  constructor(private contactService: ContactService) {}

  ngOnInit() {}

  createContact(contact: Contact) {

```

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

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

deleteContact(contactId: string): void {
  this.contactService
    .deleteContact(contactId)
    .subscribe((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">
  <app-contact-list></app-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`.

```

<link
    rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
    integrity="sha384-
Gn5384xqQ1aowXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
    crossorigin="anonymous"
/>
<script
    src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
    integrity="sha384-
KJ3o2DKtIkVYIK3UENzmM7KCKRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN"
    crossorigin="anonymous"
></script>
<script
    src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.
js"
    integrity="sha384-
ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q"
    crossorigin="anonymous"
></script>
<script
    src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"
    integrity="sha384-
JZR6Spejh4U02d8j0t6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmY1"
    crossorigin="anonymous"
></script>

```

After adding bootstrap library to our project our `src/index.html`.

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>ContactlistAngular</title>
    <base href="/" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="icon" type="image/x-icon" href="favicon.ico" />
    <link

```

```

    rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
    integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
    crossorigin="anonymous"
  />
  <script
    src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
    integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCKRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN"
    crossorigin="anonymous"
  ></script>
  <script
    src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js"
    integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q"
    crossorigin="anonymous"
  ></script>
  <script
    src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"
    integrity="sha384-JZR6Spejh4U02d8j0t6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmY1"
    crossorigin="anonymous"
  ></script>
</head>
<body>
  <app-root></app-root>
</body>
</html>

```

Finalize the deployment configuration and run the app

With our Angular code finished, we're now ready to deploy the application. We will be making the following changes in `package.json` file.

- i. Look for scripts section in the file and locate start in that.
- ii. Then change from `"start": "ng server"` to `"start": "node server.js"`.

Before change:


```
{
  "name": "mean-contactlist-angular2",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng server",
    "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"
}
}
```

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",
  "codemlizer": "~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:

```
let express = require("express");
let bodyParser = require("body-parser");
let mongoose = require("mongoose");
const contacts = require("./routers/contacts")
const path = require('path');

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

// Create link to Angular build directory
//let distDir = __dirname + "/dist/";
app.use(express.static(path.join(__dirname, './dist/contactlist-angular')));
```

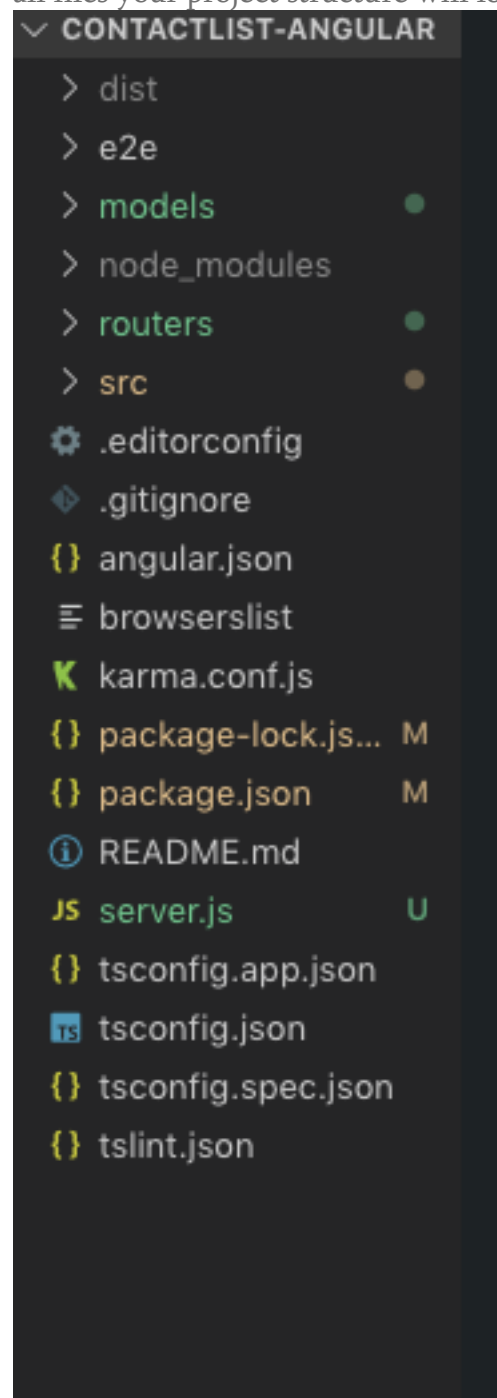
Finally, we'll modify our `app.module.ts` file to import the `HttpClientModule` and `FormsModule`:

```
import { BrowserModule } from "@angular/platform-browser";
import { NgModule } from "@angular/core";
import { FormsModule } from "@angular/forms";
import { HttpClientModule } from "@angular/common/http";
import { AppComponent } from "./app.component";
import { ContactDetailsComponent } from "./contacts/contact-details/contact-
details.component";
import { ContactListComponent } from "./contacts/contact-list/contact-
list.component";
import { ContactService } from "../app/contacts/contact.service";
@NgModule({
  declarations: [AppComponent, ContactDetailsComponent, ContactListComponent],
  imports: [BrowserModule, FormsModule, HttpClientModule],
```

```
providers: [ContactService],  
bootstrap: [AppComponent]  
}))  
export class AppModule {}
```

Final Project Structure

Add all models, routers and server.js in the mean-contactlist-angular2 folder. After adding all files your project structure will look like image below.



Now open terminal and install the following packages.

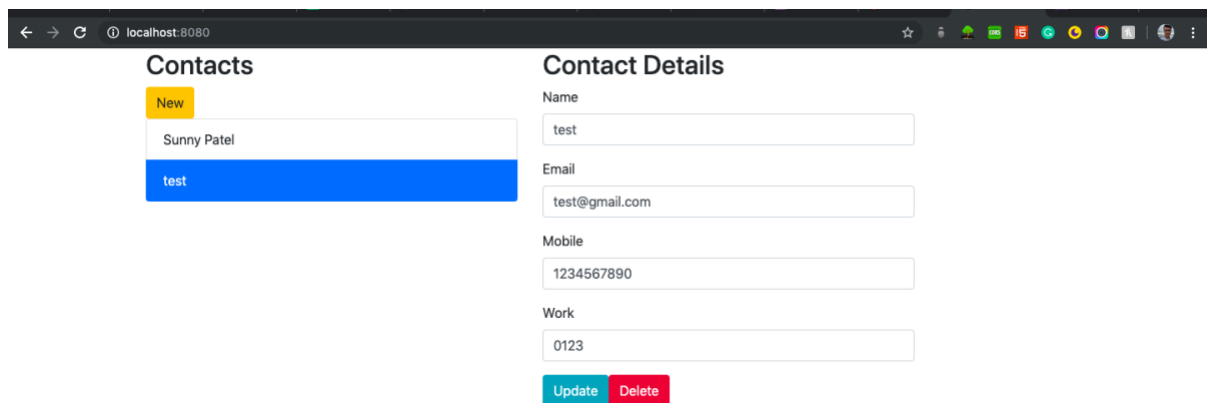
```
npm install express body-parser --save  
npm install mongoose --save
```

Complete the project and run or deploy

Now that the web application component is complete, you can view your app by opening the website from the CLI using npm start.

```
$ npm start
```

Now open the browser and <http://localhost:8080/>



The screenshot shows a web browser at localhost:8080. The page has two main panels. The left panel, titled 'Contacts', contains a yellow 'New' button and a list of two contacts: 'Sunny Patel' and 'test'. The 'test' contact is highlighted in blue. The right panel, titled 'Contact Details', contains four input fields: 'Name' (with 'test'), 'Email' (with 'test@gmail.com'), 'Mobile' (with '1234567890'), and 'Work' (with '0123'). Below these fields are two buttons: 'Update' (teal) and 'Delete' (red).

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.

References:

- <https://devcenter.heroku.com/articles/mean-apps-restful-api>
- <https://getbootstrap.com/docs/4.0/getting-started/introduction/>
- <https://github.com/sunny3p/ContactApp.git>