Build Modern App with MEAN Stack 2.0– Part 1

Pankaj Kumar Choudhary
3 May 2017 CPOL

The goal of this series to start with basic of MEAN stack development and cover all the concept that are required for MEAN stack development and finish with a web application that is completely written in JavaScript technologies.

In starting phase of JavaScript, JavaScript was only limited to the client side part of the application. We used JavaScript to handle events at client side. But last few years everything has changed, now JavaScript not used only for handle the events at client side now it is used at each phase of a software and application development for example at database level, server level etc. We can build an application that is completely written on JavaScript technologies from front end to backend and also at database level. MEAN stack application are the example of this. MEAN is an acronym for MongoDB, ExpressJS, AngularJS and Node.js. In MEAN stack we use Angular for front end, Node for server and Express as upper layer of Node and MongoDB for the database.

MEAN stack application is combination of 4 JavaScript technologies.

MongoDB

In MEAN stack "M" represents to MongoDB, MOngoDB is an open source NoSQL database that uses a document-oriented data model. In MongoDB instead of storing data in relational format like table we store data into JSON formatted.

ExpressJS

In MEAN stack "E" stands for ExpressJS, Express.js is a Node.js framework and allows JavaScript to be used outside the Web Browsers, for creating web and network applications. This means we can create the server and server-side code for an application like most of the other web languages.

AngularJS

In MEAN stack "A" stand for AngularJS, Angular is a structural framework for dynamic web apps. We use HTML as your template language and using Angular extend HTML's syntax. Latest version of Angular is 2.0 . Angular 2.0 is completely written and now it is component based. In MEAN stack 1.0 we use Angular 1.x version and for MEAN stack 2.0 we use Angular 2.0. In this series we will use Angular 2.0 for front end of application.

Node.js

In MEAN stack "N" stand for the Node.js, Node.js provide open source and cross-platform runtime environment for developing server side application. Node.js provide a platform to develop an event-driven, non-blocking I/O lightweight and efficient application.

Object of this series to provide a step by step instruction to setup the environment for MEAN stack 2.0 application and provide a deep knowledge of how MEAN stack application works.

What will we learn from this series

The goal of this series to start with basic of MEAN stack development and cover all the concept that are required for MEAN stack development and finish with a web application that is completely written in JavaScript technologies. In this series we will create "Employee Management System(EMS)", in EMS system we can manage the employee level information like project that are undertaken by employee and employee's basic details.

We will cover following topic in this series.

- Initial setup with Node.js
- Add express and make the API
- Setup database with MongoDB
- Setup front-end with Angular2
- Add a new employee entry
- View list of all employees.
- Edit the details of an existing employee
- Delete an existing employee
- Add filter and search functionality
- Add Sorting functionality
- Conclusion

Pre Requests

- Basic knowledge of Angular2, MongoDB, Node.js and Express.js. If you are fresher to all these terms then I prefer you first take a tour to all these technologies.
- Pre installation of NPM. If NPM is not configured in your system then first install NodeJS.
- Install the Visual Studio Code for the IDE, you can use any IDE but I will recommended you to use the Visual Studio Code, it makes the development easy.

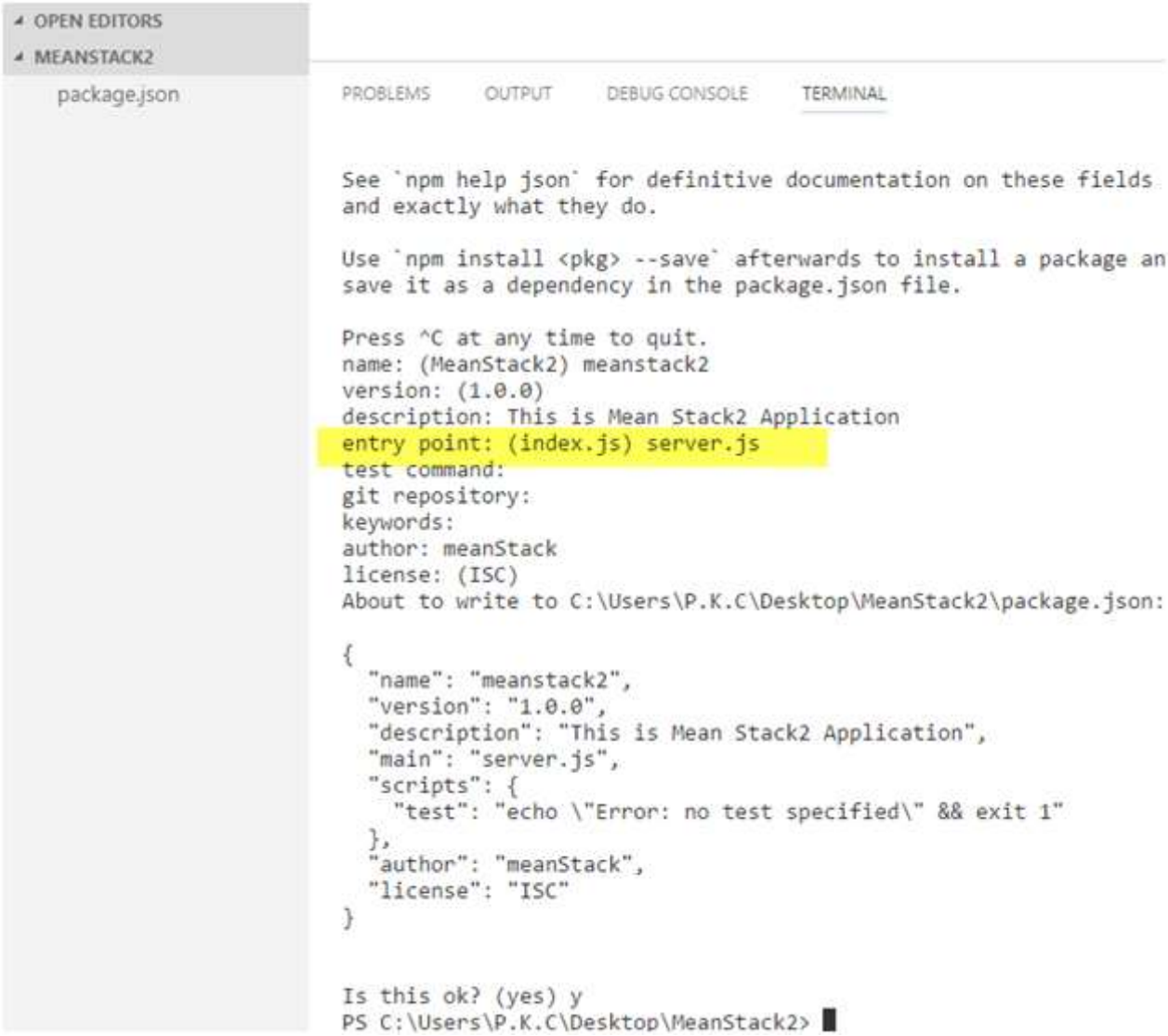
Application Setup

Now we setup development environment for our application. First of all we create server using the Node.js and Express and after that using the Angular2 we create the front end of application and use the MongoDB as our database.

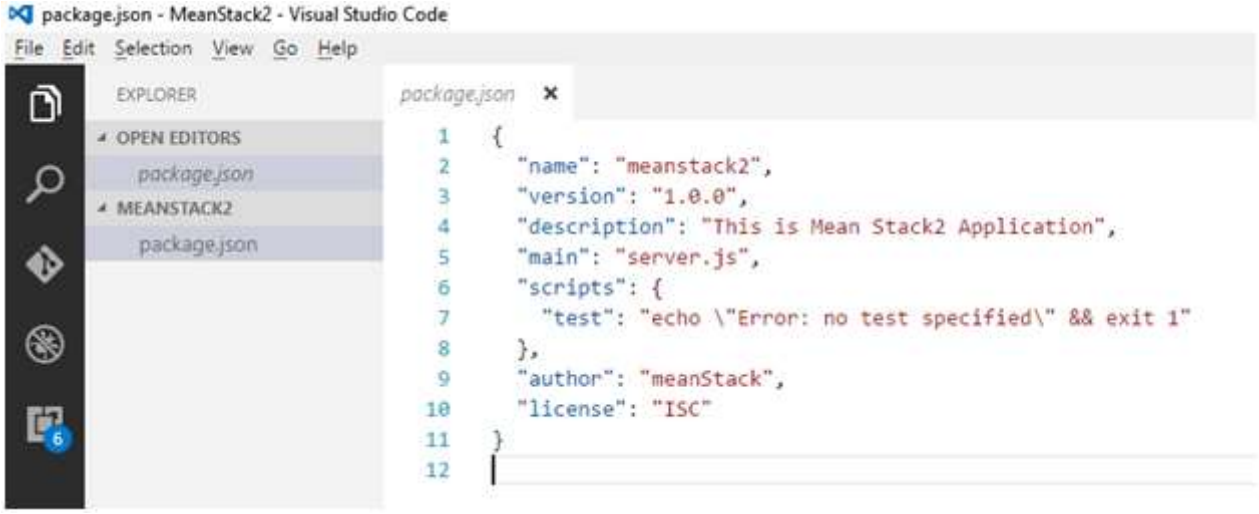
First of all create a folder anywhere in your system where you like and named this folder as "MeanStack2", now open this in Visual Studio Code. Now go to "View" tab and click on "Integrated Terminal". After open the Integrated Terminal, now follow the below steps to the application.

Step 1: Create Package.json file

In your integrated terminal paste "npm init" command and press the enter. This command will create the "*package.json*" file for Node.js application. When you run this command, it will ask you to enter some information like name, version, license etc. Point to consider that, it will ask you about the "entry point" of application, for "entry point" write the "*server.js*" instead of "*index.js*", for other fields you can enter any information that you want.

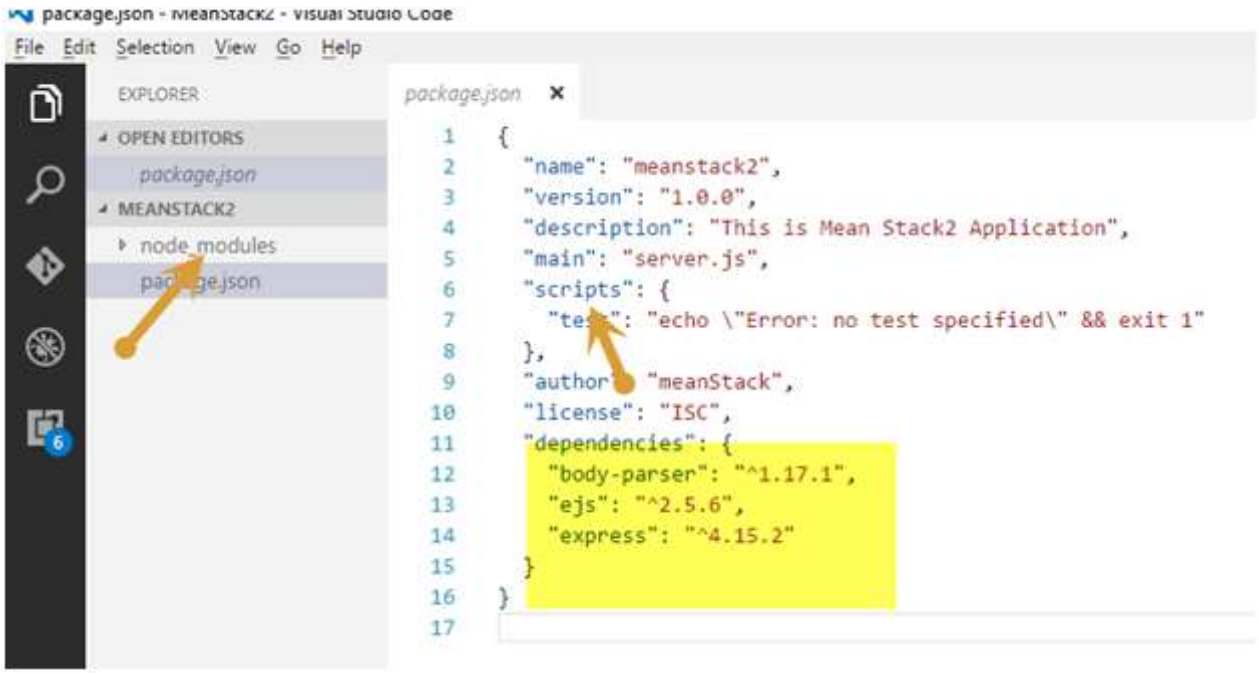


After completing this step you can see that a "*package.json*" file has been created and this file contain the information that we enter in "Integrated Terminal".



Step 2: Add express and other dependencies

For the Node.js server setup we require the some packages for express as upper layer of Node.js, ejs as template engine and body-parser to parse the request body. So paste "npm install express body-parser ejs –save" command in your integrated terminal and press enter, this command add three packages in "node_modules" folder and also add the dependency in your "package.json" file.



Step 3: Add "server.js" file

As we defined in our "*package.json*" file that "*server.js*" file is the entry point of our application, so add the "*server.js*" file in root directory of application and paste the following code into this file.

```
var express=require('express');
var path=require('path');
var bodyParser=require('body-parser');

// In above three line we import the required packages

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

```
var api=require('./routes/api');

// index and api object contain the path of routing files for our application

var port=4500;
var app=express();

//Define the port and create an object of express class

app.set('view engine','ejs');
app.set('views',path.join(__dirname,'/client/views'));

// define the view engine and set the path for views files

app.engine('html',require('ejs').renderFile);
//Register given template engine callbac function as extension

app.use(express.static(path.join(__dirname,'/client')));

// Defien the path for the static files like image, css and js files

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended:false}));
// Define the middleware to parse the data from URL request and requesy body

app.use('/',index);
app.use('/api',api);
app.use('*',index);
// define the middleware for routing

app.listen(port,function(){
    console.log('Server Started At '+port);
})
// Run the Node.js server at 4500 port
```

In above lines of code we create a server and set all the middle-ware and other required dependencies and later we run this server on "4500" port. In first three line of code we import the require modules to setup the server. In next line we import "index" and "api" files. Actually in these two files we implement the routing for our application. Later we set the "ejs" as template view engine for our application and set the path of our template pages. Using the "express.static" middle-ware function we define the path for the static files images, js and css files.

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended:false}));
```

In these two lines of code we set the bodyParser and urlencoded middle-ware. The "`bodyParser`" middle-are is used to get the data from request body like POST, PUT and DELETE type request. The "urlEncoded" middle-ware is used to get the data from URL of the request.

```
app.use('/',index);
app.use('/api',api);
app.use('*',index);
```

In above line of code we set the middle-ware for the "/" and "api" and requests. It is means if request is home page then it will be handle by code written in "*index.js*" and if request is related to the "api" then it will handled by code written in "*api.js*' file. We also define the default routing, in case of default it will be handle by the "index.js" file.

```
app.listen(port,function(){
    console.log('Server Started At '+port);
})
```

In above line of code we set the server and this server will listen to the "4500" port number.

Step 4: Add routing files

Now add the "*route*" folder to the root directory of the of application, after creating the "route" folder now add "index.js" file and paste the following code into this file.

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

router.get('/',function(req,resp,next){
    resp.render('index.html');
});

module.exports=router;
```

In above line of code we perform the routing, if request is get type then we will render the user to "*index.html*" page that is present in "*client/views*" folder. After creating the "*index.js*" page now add another page and named this page as "*api.js*", and paste the following code into this file.

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

router.get('/',function(req,resp,next){
    resp.send('this is api data');
});

module.exports=router;
```

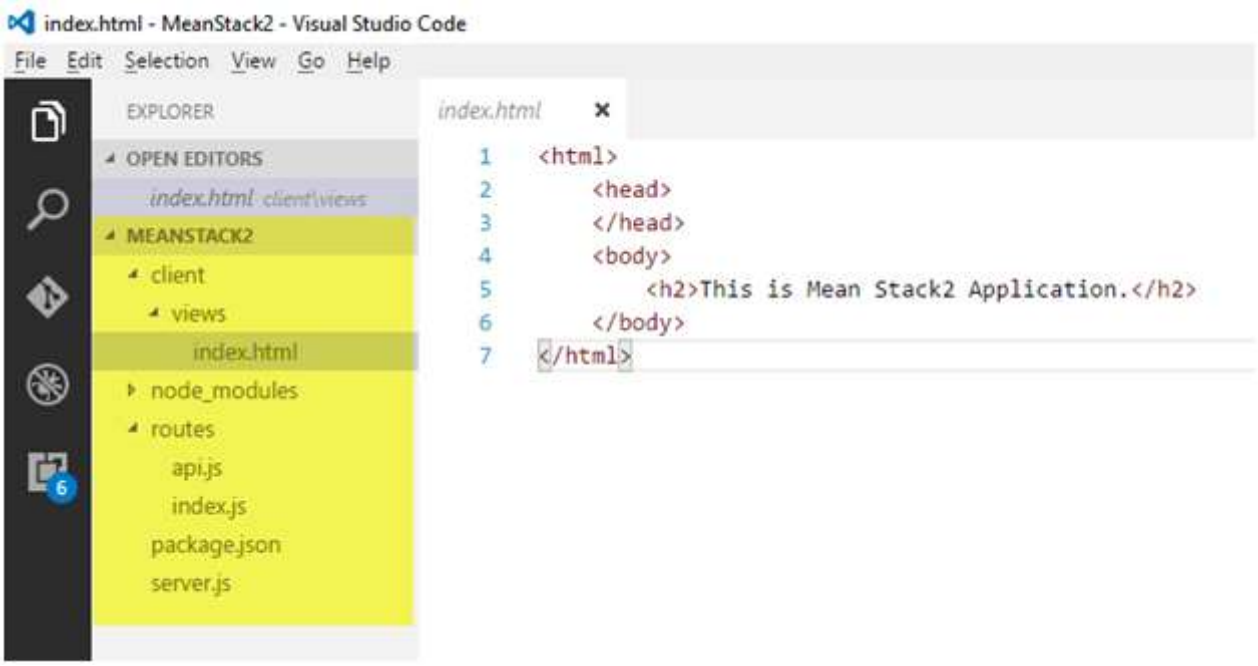
In above code we are sending a simple text message response to the request.

Step 5: Add Client folder

Now add a "*client*" folder in your application root, this folder will contain all the client side related code. After creating this file now add another folder in "*client*" folder and named this folder as "views". In "*views*" folder we will create our index.html and Angular2 related code and files. After adding the "*client*" and "*views*" folder now add "*index.html*" file in "*views*" folder and page the following code into this file.

```
<html>
    <head>
    </head>
    <body>
        <h2>This is Mean Stack2 Application.</h2>
    </body>
</html>
```

Till now we perform all the basic required configuration to run the server and check the client application. Let's check the structure of our application. Following will be the structure of our application.



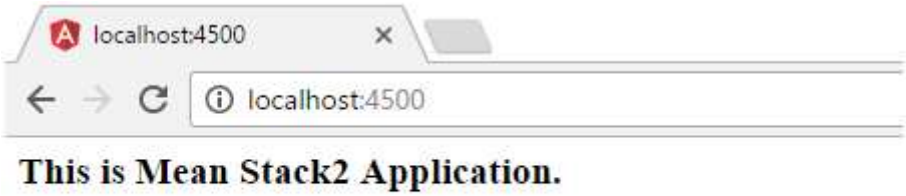
Step 6: Install the Nodemon

We can run any node.js file using the "node" command but if we make any changes in our files then we need to restart our server again and again that takes a lot of time. To overcome this scenario we can add the "nodemon", nodemon will watch the files in the directory in which nodemon was started, and if any files change, nodemon will automatically restart your node application. to add the "nodemon" package open your command line terminal and paste the "npm install –g nodemon" line of code and press the enter.

Step 7: Run the application

After adding the "nodemon" now go to "Integrated Terminal" of Visual Studio Code and run the "nodemon sever" command. After run this command if you get below screen that means everything configured very well.

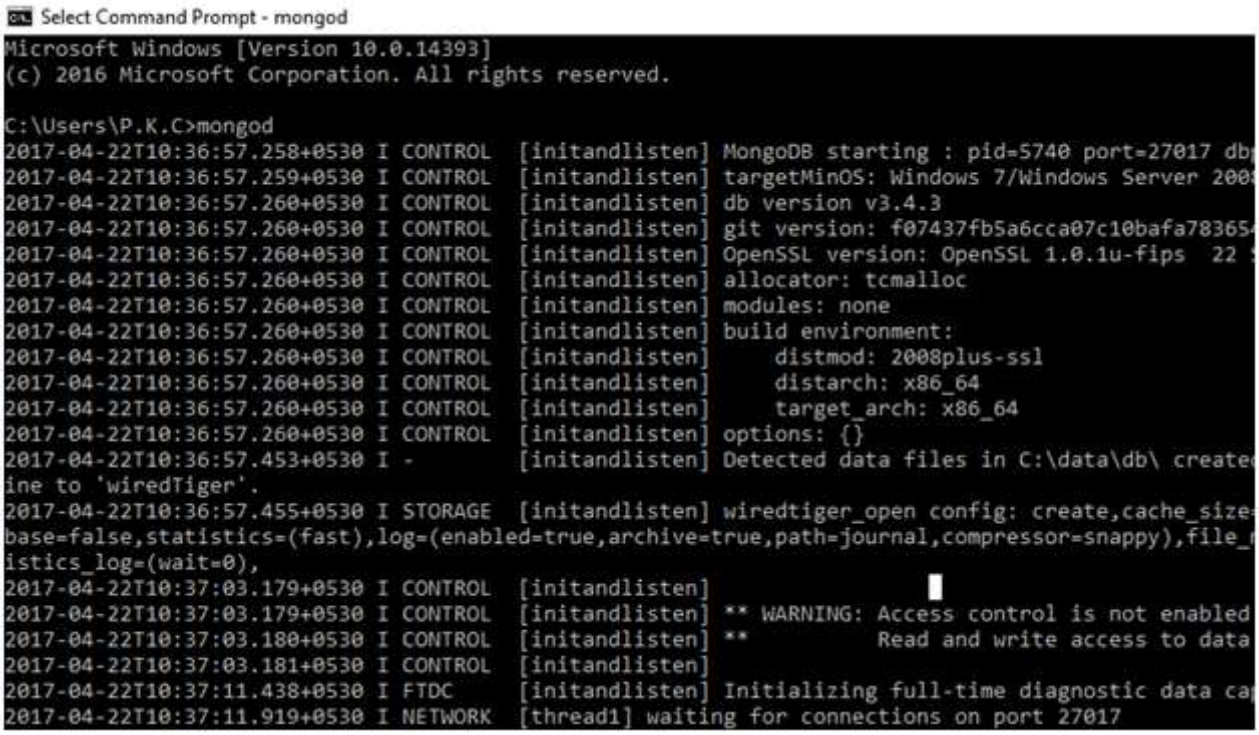
Now open your browser and paste the "localhost:4500" URL and hit enter. When press the enter and everything is configured properly then you will get this screen.



If you are not getting this screen that means something is going wrong with your code and try to find and resolve that issue. Now we create a database and establish the a connection with this database using the "mongoose" library.

Step 8: Create database and establish the connection with database

If you have installed the "MongoDB" then open a command prompt window and run the "mongod" command, this command run the "MongoDB" server at "27017" port and ready to connection with a client.



After starting the sever now we run a client and create a new database. Now open another command prompt window and run the "mongo" command, this command create a client and establish the connection to server running at port "27017".

After creating the connection to server now "`use employeeDetails`" command, this command create a new database (employeeDetails) and switched to this new database.

Now paste the following code into client command prompt and press the enter.

```
db.employees.insert([{"EmployeeName":"Ankur Verma","Designation":"Mobile
Developer","Project":"OUP","Skills":"Java, Android Studio, Xamarin"},{"EmployeeName":"Dheeraj
Sharma","Designation":" Developer","Project":"Lion Servcies","Skills":"C#,Asp.Net,MVC,AngularJS"},
{"EmployeeName":"Dhramveer","Designation":"
Developer","Project":"VMesh","Skills":"C#,Asp.Net,MVC,AngularJS"},
{"EmployeeName":"Prakash","Designation":" Web
Developer","Project":"OUP","Skills":"C#,Asp.Net,MVC,AngularJS"},
{"EmployeeName":"Raj Kumar","Designation":"Developer","Project":"CNS","Skills":"C#,Asp.Net,MVC"}])
```

Above code create a new collection in our database and insert some data into this collections. After inserting some default data into "employees" collection' let's check the data that we entered. To get the data from a collection we run the "find" command.

Now paste "`db.employees.find().pretty()`" command and press the enter. When you run this command it will show all the data of "employees" collection.

Now our database is ready let's create model classes and make a connections to this database.

Step 9: Install Mongoose and make connection to the database

Now open "Integrated Terminal" of Visual Studio Code and run the "npm install mongoose --save" command, this command will install the dependencies for the mongoose. In this project we are going to use the "mongoose" for MongoDB CRUD operation. The mongoose is an object modeling package for Node that essentially works like an ORM that you would see in other languages (Entity framework for C#). Using Mongoose, we can define the schema for the documents in a particular collection. It provides a lot of convenience in the creation and management of data in MongoDB.

After installing the dependency for mongoose now create a "database" folder in root directory of the project and create a "*dataFile.js*" file in this folder and paste the following code into this file.

```
var mongoose = require('mongoose');
// Connection URL
var db = 'mongodb://localhost:27017/employeeDetails';
    // Use connect method to connect to the Server
 mongoose.connect(db, function (error) {
    if (error) {
```

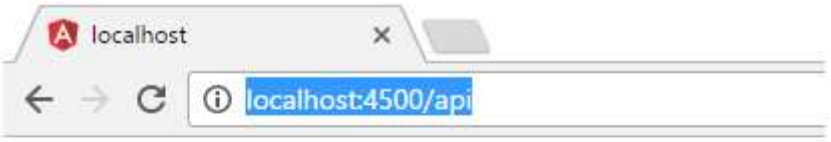
```
        console.log(error);
    }
});

var Schema = mongoose.Schema;
var Employee_Schema = new Schema({
    EmployeeName: String,
    Designation: String,
    Project: String,
    Skills:String
});

var Employee = mongoose.model('employees', Employee_Schema);

 module.exports=Employee;
```

In first lines of code we simply make a connection to our MongoDB database. We create a schema for the "employees" collection and define totally four attribute for the "Employee_Schema" and in next line we create an "employees" model using this schema. So when we perform any CRUD operation we will take this employee model and perform the CRUD operations.

After creating the Employee models and schema now go to "*api.js*" file and replace the code of this file with following code.

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

var Employee=require('../database/dataFile');

router.get('/',function(req,resp,next){
    Employee.find({},function(err,docs){
        resp.send(docs);
    })
});

 module.exports=router;
```

In above code we import the "Employee" model that we created in "dataFile.js" file and execute the "Find" method, this "find" method is similar to the "find" method of the MongoDB, so when we hit this api it will fetch out the all record form "Employees" collection and return this data as response. Let's check that, it is working or not. Now save all the changes and open you browser and paste this URL "http://localhost:4500/api" in browser and press the enter. When we hit the "api/" route it will return the list of all employees as below.




```
[
  - {
        _id: "58fbf3a20fdd9365fbb555c3",
        EmployeeName: "Ankur Verma",
        Designation: "Mobile Developer",
        Project: "OUP",
        Skills: "Java, Android Studio, Xamarin"
    },
  - {
        _id: "58fbf3a20fdd9365fbb555c4",
        EmployeeName: "Dheeraj Sharma",
        Designation: " Developer",
        Project: "Lion Servcies",
        Skills: "C#,Asp.Net,MVC,AngularJS"
    },
  - {
        _id: "58fbf3a20fdd9365fbb555c5",
        EmployeeName: "Dhramveer",
        Designation: " Developer",
        Project: "VMesh",
        Skills: "C#,Asp.Net,MVC,AngularJS"
    },
  - {
        _id: "58fbf3a20fdd9365fbb555c6",
        EmployeeName: "Prakash",
        Designation: " Web Developer",
        Project: "OUP",
        Skills: "C#,Asp.Net,MVC,AngularJS"
    },
  - {
        _id: "58fbf3a20fdd9365fbb555c7",
        EmployeeName: "Raj Kumar",
        Designation: "Developer",
        Project: "CNS",
        Skills: "C#,Asp.Net,MVC"
    }
]
```


Now our "API", "Database" and "Server" all are ready, only remaining part is "client" application. Let's start the work on "client" application of the project.

Step 10: Create package.json file

Now go to "*client*" folder and create a "*package.json*" file and past the below code into this file.

```
{
  "name": "mytasklist",
  "version": "1.0.0",
  "scripts": {
    "start": "concurrently \"npm run tsc:w\" \"npm run lite\" ",
    "lite": "lite-server",
    "tsc": "tsc",
```

```
    "tsc:w": "tsc -w"
  },
  "licenses": [
    {
      "type": "MIT",
      "url": "https://github.com/angular/angular.io/blob/master/LICENSE"
    }
  ],
  "dependencies": {
    "@angular/common": "~2.1.1",
    "@angular/compiler": "~2.1.1",
    "@angular/core": "~2.1.1",
    "@angular/forms": "~2.1.1",
    "@angular/http": "~2.1.1",
    "@angular/platform-browser": "~2.1.1",
    "@angular/platform-browser-dynamic": "~2.1.1",
    "@angular/router": "~3.1.1",
    "@angular/upgrade": "~2.1.1",
    "angular-in-memory-web-api": "~0.1.13",
    "bootstrap": "^3.3.7",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.8",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.39",
    "zone.js": "^0.6.25"
  },
  "devDependencies": {
    "@types/core-js": "^0.9.34",
    "@types/node": "^6.0.45",
    "angular-cli": "^1.0.0-beta.28.3",
    "concurrently": "^3.0.0",
    "lite-server": "^2.2.2",
    "typescript": "^2.0.3"
  }
}
```

The above code define all the packages and dependencies that we need to create a Angular2 client and also contain the commands for start, typescript, and lite server. lite-server provide a Lightweight development only node server that serves a web app, opens it in the browser, refreshes when html or javascript change, injects CSS changes using sockets, and has a fallback page when a route is not found.

Step 11: Create tsconfig.json file

Now create another file in client folder and named this file as "*tsconfig.json*" and paste the following code into this file. This file contains the some configuration code for the typescript.

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  }
}
```

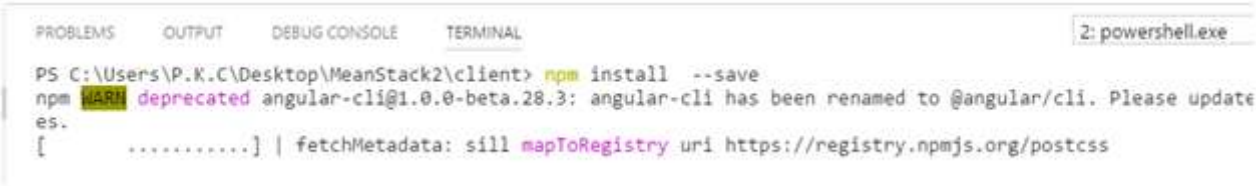
Step 12: Create systemjs.config.js file

Again create a new file in client folder and named this file as "*systems.config.js*" and paste the following code into this file. This file contain all information about the client system like angular packages, path and main file for the application(main.js) and also contain other informations.

```
/**
 * System configuration for Angular samples
 * Adjust as necessary for your application needs.
 */
(function (global) {
  System.config({
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      app: 'app',
      // angular bundles
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-browser.umd.js',
      '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-dynamic/bundles/platform-
browser-dynamic.umd.js',
      '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
      '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
      '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
      // other libraries
      'rxjs':                      'npm:rxjs',
      'angular-in-memory-web-api': 'npm:angular-in-memory-web-api',
    },
    // packages tells the System loader how to load when no filename and/or no extension
    packages: {
      app: {
        main: './main.js',
        defaultExtension: 'js'
      },
      rxjs: {
        defaultExtension: 'js'
      },
      'angular-in-memory-web-api': {
        main: './index.js',
        defaultExtension: 'js'
      }
    }
  });
})(this);
```

Step 13: Install all the packages

We defined a list of the packages in "package.json" file, so now we install all these packages. For this move to your "client" directory and run the "npm install --save" command this command take a little bit time and install all the packages that we defined in "package.json" file. After completion of this command you will find that a new folder "node_modules" has been created and this folder contains all the required modules.



Step 14: Create app.module.ts file

Now create a folder in your client directory and named this folder as "app". After creating the folder now create "*app.module.ts*" file and paste the following code into this file. The app.module.ts file is parent module of all other modules, when ever we create a component, services, filter and directives then first of all we need to register all of these in app.module.ts file.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpModule } from '@angular/http';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './components/app/app.component';
@NgModule({
    imports: [BrowserModule, HttpModule, FormsModule],
    declarations: [AppComponent,
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 15: Create Main.ts file

Now create a new file in client directory and named this file as "*main.ts*" and paste the following code into this file. This file is starting point of our application and bootstrap the AppModule.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

Step 16: Create app component

Now create an "app" folder in client directory. In this app folder we will insert all our components, services and custom filters. After creating the "app" folder now create another folder in this "App" folder and named this folder as "components", in this "component" folder we create the components. After creating the "components" again create an "app" folder into this "components" folder, in this "app" folder we create files for the "app" component.

Now create an "app.component.ts" file and paste the following code into this file.

```
import { Component } from '@angular/core';
@Component({
  moduleId: module.id,
  selector: 'my-app',
  templateUrl: 'app.component.html'

})
export class AppComponent { }
```

In above code we create a simple component and defined the app.component.html" as templateUrl for this component, so we need to create an html template file also. Again create a file and named this file as "app.component.html" and paste the following code into this file.

```
<div class="container">
    <h1>Congrats! You set MEAN Stack2 Application Successfully</h1>
</div>
```

Till now we create all the files and make all the setup that are required to run a MEAN Stack application, a last step is reaming to run the application that start the client app. Now open "integrated Terminal", moves to "client" directory and run the "npm start" command. This command build all the typescript file and create the corresponding JavaScript and map files. The "npm start" command run the "lite server".

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\P.K.C\Desktop\MeanStack2\client> npm start

> mytasklist@1.0.0 start C:\Users\P.K.C\Desktop\MeanStack2\client
> concurrently "npm run tsc:w" "npm run lite"

[0]
[0] > mytasklist@1.0.0 tsc:w C:\Users\P.K.C\Desktop\MeanStack2\client
[0] > tsc -w
[0]
[1]
[1] > mytasklist@1.0.0 lite C:\Users\P.K.C\Desktop\MeanStack2\client
[1] > lite-server
[1]
[1] Did not detect a `bs-config.json` or `bs-config.js` override file. Usi
[1] ** browser-sync config **
[1] { injectChanges: false,
[1]   files: [ './**/*.{html,htm,css,js}' ],
[1]   watchOptions: { ignored: 'node_modules' },
[1]   server: { baseDir: './', middleware: [ [Function], [Function] ] } }
[1] [BS] Access URLs:
[1]  ---------------------------------------
[1]        Local: http://localhost:3000
[1]     External: http://192.168.42.1:3000
[1]  ---------------------------------------
[1]           UI: http://localhost:3001
[1]  UI External: http://192.168.42.1:3001
[1]  ---------------------------------------
[1] [BS] Serving files from: ./
[1] [BS] Watching files...
[1] 17.04.26 07:55:16 404 GET /index.html
[1] [BS] Reloading Browsers...
```

If your "node" server already running then refresh the browser else run the "nodemon server" command and you will get the following output on your screen.

If you get the above screen then congratulation to you, here we complete the MEAN Stack configuration and now we start the work on our "Employee Management" system part. If you don't get above screen then there some chances that you are getting some error, so try to resolve that error.

Add bootstrap and create project layout

Till now we successfully configured the MEAN Stack 2 application, now we add some required files for the bootstrap and create the layout for our application. Now go to "index.html" file add the following code at top of the head section. In below code we add CDN for the jQuery and bootstrap and we also add a "base" tag in the first line, this base tag will be use at the time of routing and provide a base(common) URL to all the routes that we will create later in this project.

```
<base href="/" />
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

<!-- jQuery library -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script>

<!-- Latest compiled JavaScript -->
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
```

Create Home component

We create a home component and show the list of all employees using this component. Now go to component folder add a new folder and named this folder as "home", in this folder now add a new "*home.component.ts*" file and paste following code.

```
import { Component } from '@angular/core';

@Component({
    moduleId: module.id,
    selector: 'home',
    templateUrl: 'home.component.html',

})
export class homeComponent {
}
```

Now create a "*home.component.html*" file and paste following code into this file.

```
<h1>This is home page</h1>
```

Add newEmployee component

Using this component we will provide the functionality to add a new employee. Now go to component folder add a new folder and named this folder as "newEmployee", in this folder now add a new "*newEmployee.component.ts*" file and paste following code.

```
import { Component } from '@angular/core';

@Component({
    moduleId: module.id,
    selector: newEmployee,
    templateUrl: newEmployee.component.html',

})
```

```
export class newEmployee Component {
}
```

Let's add a template file for this component, now create a "*newEmployee.component.html*" file and paste following code into this file.

```
<h2>Add New Employee</h2>
```

Add details Component

Using the "details" component we will show the details of an employee. similar to previous step add a "details" folder and add "Details.component.ts" file and paste the following code into this file.

```
import { Component } from '@angular/core';

@Component({
    moduleId: module.id,
    selector: 'edit',
    templateUrl: 'edit.component.html',

})
export class editComponent {
}
```

Now create a "details.component.html" page and paste the following code into this file.

```
<h2>This is a Details page</h2>
```

Add edit component

Using this component we add the functionality to edit the information of an existing employee. So create a "edit" folder and create "Edit.component.ts"file and paste following code into this file.

```
@Component({
    moduleId: module.id,
    selector: 'nav-menu',
    templateUrl: 'navmenu.component.html',
    styleUrls: ['navmenu.component.css']
})
export class NavMenuComponent {
}
```

Now create "edit.component.html" page and paste the following code.

<h1>This is edit page</h1>

Create navMenu Component

We will add a side menu into our application and using this side menu we will provide the functionality of redirect to the "newEmployee" and "Home" page of the application. So add a "navmenu" folder into component section and create a "*navmenu.component.ts*" file and paste the following code.

```
import { Component } from '@angular/core';

@Component({
```

```
    moduleId: module.id,
    selector: 'nav-menu',
    templateUrl: 'navmenu.component.html',
    styleUrls: ['navmenu.component.css']
})
export class NavMenuComponent {
}
```

Now create a "*navmenu.component.html*" file and paste the following code.

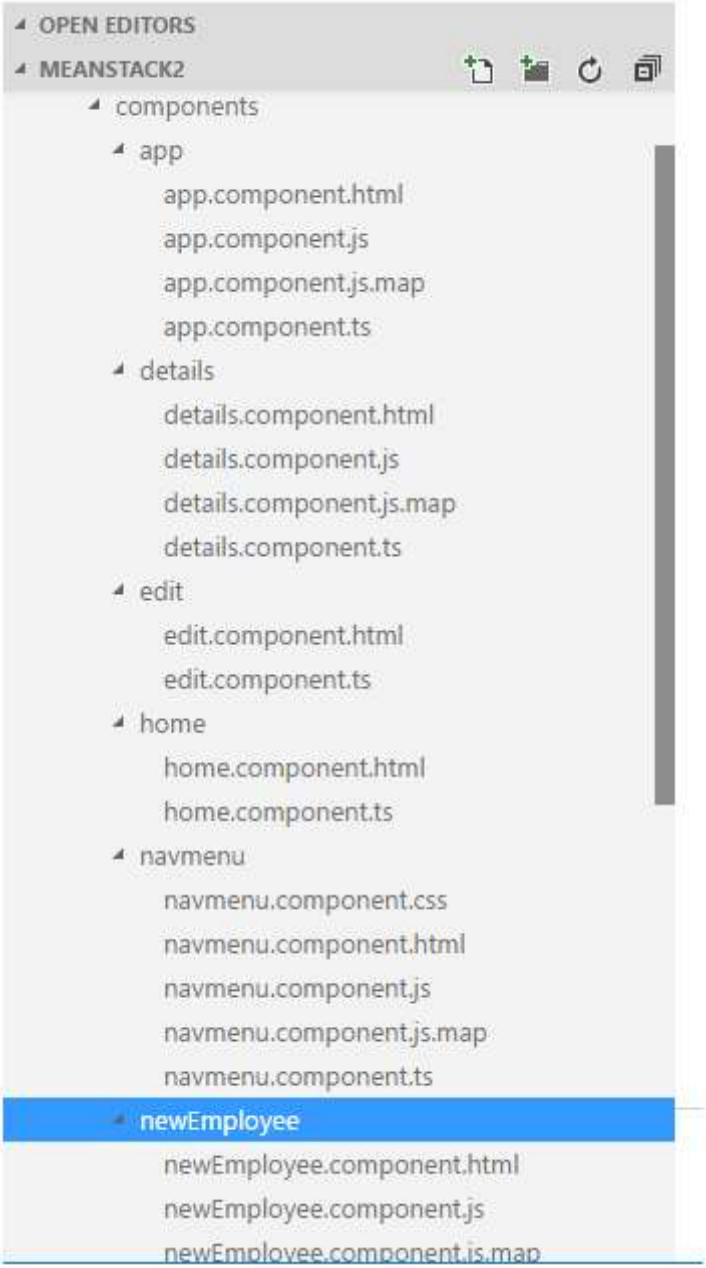
```
<div class='main-nav'>
    <div class='navbar navbar-inverse'>
        <div class='navbar-header'>
            <button type='button' class='navbar-toggle' data-toggle='collapse' data-
target='.navbar-collapse'>
                <span class='sr-only'>Toggle navigation</span>
                <span class='icon-bar'></span>
                <span class='icon-bar'></span>
                <span class='icon-bar'></span>
            </button>
            <a class='navbar-brand' [routerLink]="['/home']">EMS</a>
        </div>
        <div class='clearfix'></div>
        <div class='navbar-collapse collapse'>
            <ul class='nav navbar-nav'>
                <li [routerLinkActive]="['link-active']">
                    <a [routerLink]="['/home']">
                        <span class='glyphicon glyphicon-home'></span> Home
                    </a>
                </li>
                <li [routerLinkActive]="['link-active']">
                    <a [routerLink]="['/new']">
                        <span class='glyphicon glyphicon-user'></span> New Employee
                    </a>
                </li>
            </ul>
        </div>
    </div>
</div>
```

After creating the ".ts" and ".html" file now we create a ".css" file to add the style for this sidemenu, so add the "sidemenu.component.css" file and paste the following code into this file.

```
li .glyphicon {
    margin-right: 10px;
}

/* Highlighting rules for nav menu items */
li.link-active a,
li.link-active a:hover,
li.link-active a:focus {
    background-color: #4189C7;
    color: white;
}

/* Keep the nav menu independent of scrolling and on top of other items */
.main-nav {
    position: fixed;
    top: 0;
    left: 0;
    right: 0;
    z-index: 1;
}
```

```
@media (min-width: 768px) {
    /* On small screens, convert the nav menu to a vertical sidebar */
    .main-nav {
        height: 100%;
        width: calc(25% - 20px);
    }
    .navbar {
        border-radius: 0px;
        border-width: 0px;
        height: 100%;
    }
    .navbar-header {
        float: none;
    }
    .navbar-collapse {
        border-top: 1px solid #444;
        padding: 0px;
    }
    .navbar ul {
        float: none;
    }
    .navbar li {
        float: none;
        font-size: 15px;
        margin: 6px;
    }
    .navbar li a {
        padding: 10px 16px;
        border-radius: 4px;
    }
    .navbar a {
        /* If a menu item's text is too long, truncate it */
        width: 100%;
        white-space: nowrap;
        overflow: hidden;
        text-overflow: ellipsis;
    }
}
```

After adding all these component now following will be the structure of our "component" section.



Register the components and perform the routing

After creating all the required components now we needs to register all these components into "*app.modules.ts*" file, so open your "*app.modules.ts*" file and replace the code with following code into this file.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpModule } from '@angular/http';
import { FormsModule } from '@angular/forms';
import { RouterModule } from '@angular/router';
import { AppComponent } from './components/app/app.component';
import {NavMenuComponent}  from './components/navmenu/navmenu.component';
import { newEmployeeComponent } from "./components/newEmployee/newEmployee.component";
import {homeComponent}  from "./components/home/home.component";
import {editComponent} from "./components/edit/edit.component";
import {detailsComponent} from "./components/details/details.component";
@NgModule({
    declarations: [
    AppComponent,
```

```
    NavMenuComponent,
    newEmployeeComponent,
    homeComponent,
    editComponent,
    detailsComponent
    ],
    imports: [BrowserModule,
     HttpModule,
    FormsModule,
    RouterModule.forRoot([
            { path: '', redirectTo: 'home', pathMatch: 'full' },
            { path: 'home', component: homeComponent },
            { path: 'details/:id', component: detailsComponent },
            { path: 'new', component: newEmployeeComponent },
            { path: 'edit/:id', component: editComponent },
            { path: '**', redirectTo: 'home' }
        ])],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

In above code lines of code we register all the components into "declarations" section and also perform the routing for these components.

Add router-outlet

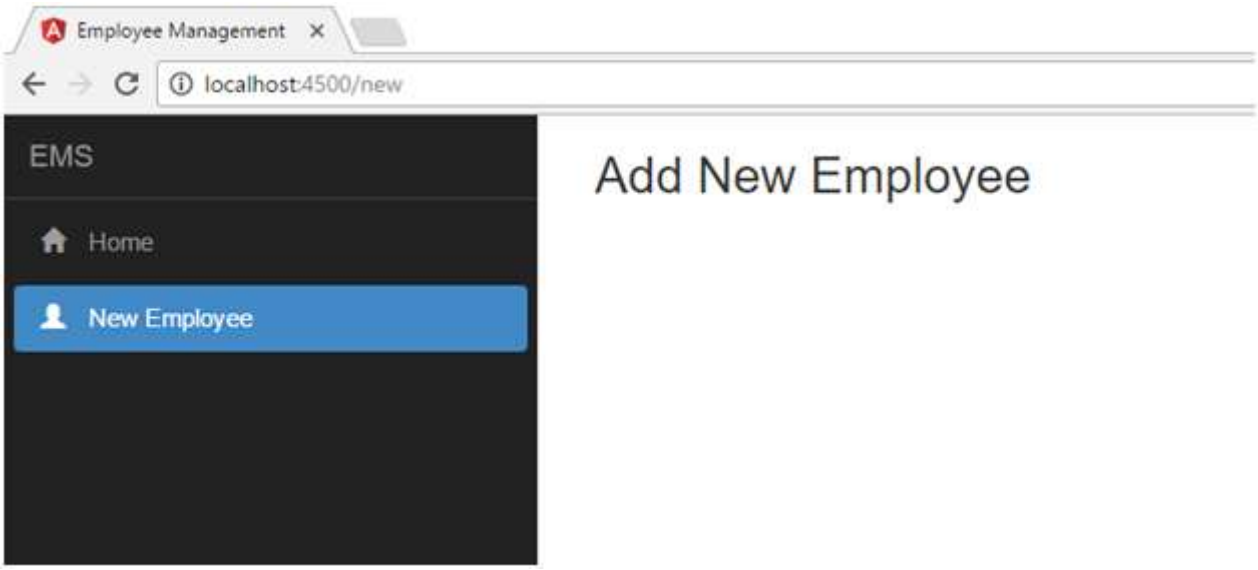
Now go to your "app.component.html" page and replace the code of this file with following code.

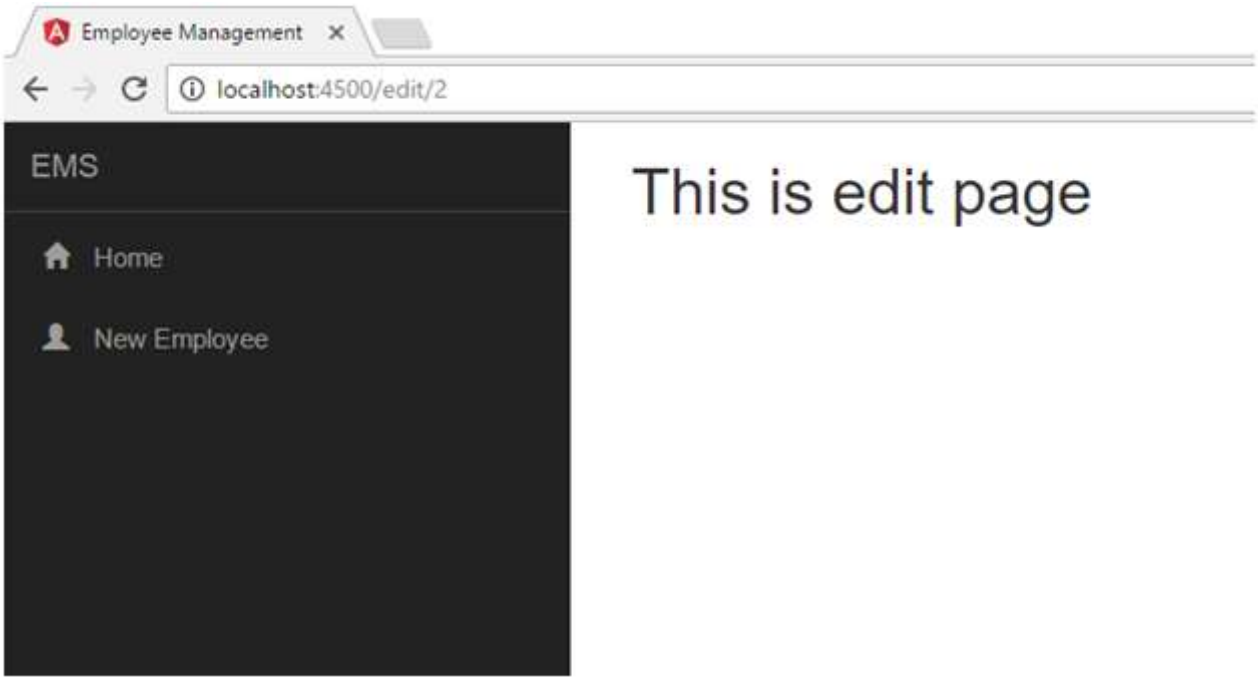
```
<div class='container-fluid'>
    <div class='row'>
        <div class='col-sm-3'>
            <nav-menu></nav-menu>
        </div>
        <div class='col-sm-9 body-content'>
           <router-outlet></router-outlet>
        </div>
    </div>
</div>
```

In above code we add the "sidemenu" using the "nav-menu" selector and also add the "router-outlet", this router-outlet work as a primary outlet for all the components, in other words `RouterOutlet` is a "placeholder" component that gets expanded to each route's content.

After making all the changes now save all the changes and refresh or restart your application. Now following will be the structure of our application.


Employee Management
localhost:4500/home
EMS
Home
New Employee
This is home page


Employee Management
localhost:4500/new
EMS
Home
New Employee
Add New Employee


Employee Management
localhost:4500/edit/2
EMS
Home
New Employee
This is edit page

Add service

Now, our API is ready to return the employee list. Let's use this API and display the employee information. We need a Service, which can use this API and get the result. Add a Service folder in an app section. After creating the folder, add a TypeScript file and name this file as services.ts and paste the code given below into this file.

```
import { Injectable } from '@angular/core';
import { Http, Headers, RequestOptions, Response } from '@angular/http';
import { Observable } from "RxJS/Rx";

@Injectable()
export class EmployeeServcies {
    constructor(private http: Http) {

    }
    getEmployeeList() {
        return this.http.get('http://localhost:4500/api');
    }

}
```

In the code given above, we create an `EmployeeServcies` class and decorate this with @Injectable meta data. "@Injectbale meta data is used to define a class as a Service. In this Service class, we add `getEmployeeList` method. In this method, we are using GET method of HTTP class to perform HTTP GET request to the Server.

After creating the Service, now we need to register this Service in *app.modules.ts* file. Open your app.modules.ts file, import this Service and register this Service into providers. We are registering this Service in an *app.modules.ts* file, which means now this Service is a global Service. We can use this Service in any components and we don't need to register this Service in each component.

```
import { RouterModule } from '@angular/router';
import { AppComponent } from './components/app/app.component'
import {NavMenuComponent}  from './components/navmenu/navmenu.component';
import { newEmployeeComponent } from "./components/newEmployee/newEmployee.co
import {homeComponent}  from "./components/home/home.component";
import {editComponent} from "./components/edit/edit.component";
import {detailsComponent} from "./components/details/details.component";
import {EmployeeServcies} from "./services/services" ;
@NgModule({
    declarations: [
    AppComponent,
    NavMenuComponent,
    newEmployeeComponent,
    homeComponent,
    editComponent,
    detailsComponent
    ],
    providers:[EmployeeServcies],
    imports: [BrowserModule,
     HttpModule,
    FormsModule,
    RouterModule.forRoot([
            { path: '', redirectTo: 'home', pathMatch: 'full' },
            { path: 'home', component: homeComponent },
            { path: 'details/:id', component: detailsComponent },
            { path: 'new', component: newEmployeeComponent },
            { path: 'edit/:id', component: editComponent },
            { path: '**', redirectTo: 'home' }
        ])],
```

link-active

After creating and registering the Service, now we use this Service in our home component, so open *home.component.ts* file and paste the code given below.

```
import { EmployeeServcies } from './../../services/services';
import { Component } from '@angular/core';
import { Response } from '@angular/http';

@Component({
    moduleId: module.id,
    selector: 'home',
    templateUrl: 'home.component.html',

})
export class homeComponent {
      public EmployeeList = [];
    public constructor(private empService: EmployeeServcies) {
        this.empService.getEmployeeList()
            .subscribe(
            (data: Response) => (this.EmployeeList = data.json())
            );

    }
}
```

In the code given above, we create an instance(empService) of EmployeeServcies Service and use the `getEmployeeList` method of this Service to get the employee list. You can see that we are using a subscribe. Actually, an Angular 2 provides a new pattern to run asynchronous requests, which are knojwn as Observables, http is the successor to Angular 1's `$http.` Instead of returning a Promise,

its `http.get()` method returns an Observable object. Using subscribe, method we can use the observable, the "subscribe" method tells the observable that you perform your task here is some one who listening and watching you. Perform the callback function when you return the result. In subscribe method, we get the data and assign the data into EmployeeList. Now, we use this list to display the employee list.

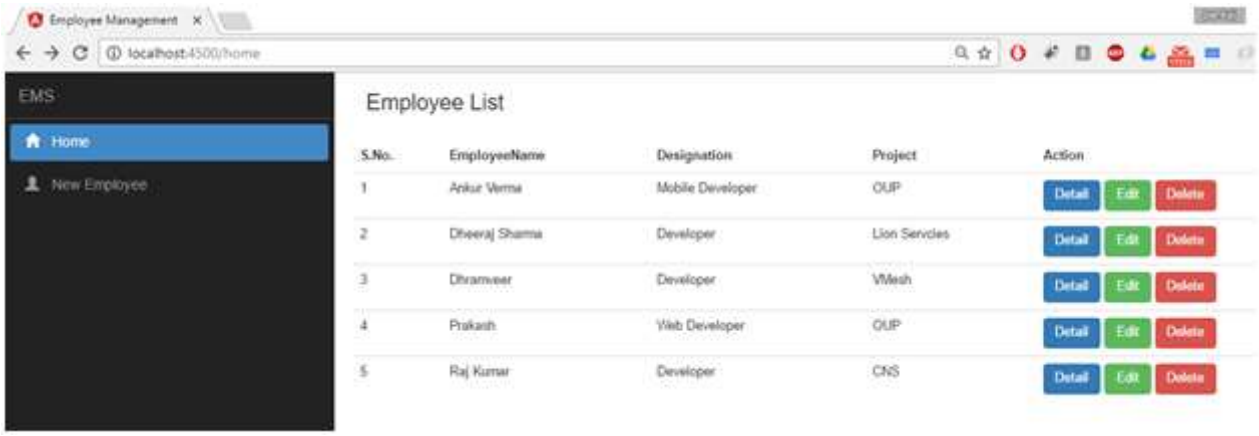
Open *home.component.html* file and paste the code given below.

```
<div class="row">
    <div class="col-md-12">
        <h3>Employee List</h3>
        <br />

    </div>
</div>
<div class="row">
    <div class="table-responsive">
        <table class="table">
            <thead>
                <tr>
                    <th>

                        S.No.
                    </th>
                    <th>
                        EmployeeName
                    </th>
                    <th>
                        Designation
                    </th>
                    <th>
                        Project
                    </th>
                    <th>
                        Action
                    </th>
                </tr>
            </thead>
            <tbody>

                <tr *ngFor="let empData of EmployeeList  ; let i = index; trackBy: employeeId">
                    <td>
                        {{i+1}}
                    </td>
                    <td>
                        {{empData.EmployeeName}}
                    </td>
                    <td>
                       {{empData.Designation}}
                    </td>
                    <td>
                       {{empData.Project}}
                    </td>
                    <td>

                        <a [routerLink]="['/details/',empData._id]"
                           class="btn btn-primary">
                            Detail
                        </a>
                        <a [routerLink]="['/edit/',empData._id]"
                           class="btn btn-success">
                            Edit
                        </a>
                        <a
                           class="btn btn-danger" (click)="deleteEmployee(empData._id)">
```

```
                        Delete
                    </a>
                </td>
            </tr>

        </table>
    </div>
</div>
```

In the code given above, we perform ngFor on an EmployeeList and create a list of employees. We also add three anchor tags (Edit, Delete and Detail) for an each employee entry. We use the routerLink to link the anchor tag to the specific part of our app.

After making all the changes, save the project and refresh the Browser and you will get the result given below.



Summary

This is the first part of "Build Modern App with MEAN Stack 2.0" series. Today, we learned how to setup Angular 2 Application with Node.js. Create a Node.js server, use the mongoose library to make the connection with MongoDB database. We created some component and performed the routing for these components. Create Service to get the data from Web API, using builtin http Service and display these data into a tabular format. In the next article, we will create some screens to add, delete or edit the employee information. We also provide the functionality of searching and sorting employee information.

You can download this project from the https://github.com/Pankajmalhan/MeanStack2

License

About the Author

Pankaj Kumar Choudhary

Student

India

Pankaj Kumar Choudhary loves Microsoft and Database technologies. He has experience on several database technology like SQL Server, MySQL, Oracle, MongoDB, PostgreSQL . He has knowledge of several technology like Asp.Net MVC, Entity Framework , Android, Php, AngularJS, Node.js, Angular2, React, Ionic and Android.

Comments and Discussions

6 messages have been posted for this article Visit **https://www.codeproject.com/Articles/1185294/Build-Modern-App-with-MEAN-Stack-Part** to post and view comments on this article, or click **here** to get a print view with messages.