Enhancing the App -

Routing and Edit Posts

[Enhancing the app 2](#_Toc168025130)

[1. Adding routing to the app 2](#_Toc57364105)

[2. Routes 4](#_Toc2045838879)

[3. Styling links 7](#_Toc1053367236)

[3. Client side vs server side routing 8](#_Toc87949584)

[4. Creating the edit form 8](#_Toc699909152)

[5. Updating posts on the server 13](#_Toc1605183045)

[6. Re-organizing backend route 14](#_Toc1621863701)

[7. Adding Loading spinners 17](#_Toc653291336)

[Appendix 2](#_Toc1931080688)[2](#_Toc752766105)

# 

# Enhancing the app

By now we have a basic MEAN app up and running with Angular frontend, node/express as the backend service and mongo as our data store.

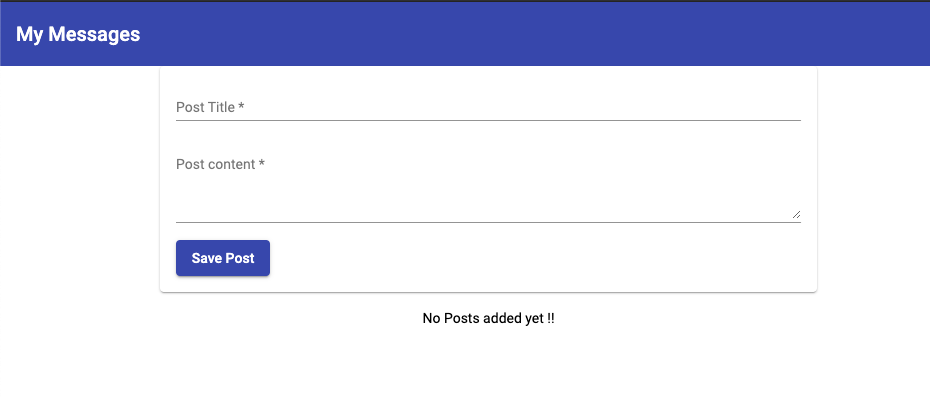
So far, we have only scratched the surface and there are other important features that we must understand, like better error handling mechanisms, securing the app and so on.

We will now enhance our app. We need to add support to edit posts and also add few other pages within the app. Adding pages does not mean that this would be a multipage app. But several pages/components rendered within the same page which gives the perception of a web application with multiple pages

Let us now start adding multiple pages. The Angular feature that would help us with that is angular router.

## 1. Adding routing to the app

The app at this point should look like below.



It contains a simple page which provides an option to add posts and list the added posts.

It does not have header links/tab like structure or other pages where we can navigate.

We can now start creating and adding separate “pages”. (Note the quote - we are not adding new HTML pages, but adding new sections. Routing will give us a perception of navigating between pages)

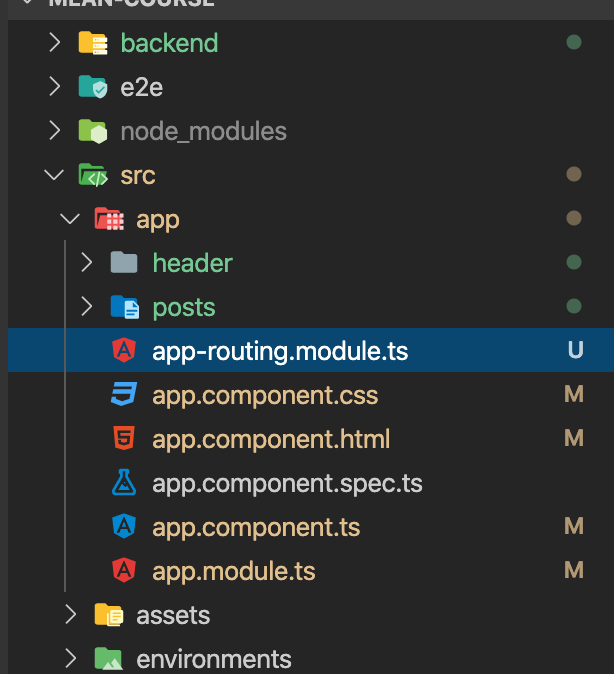
1. We can have a main page to list the posts.
2. The section where we add posts, we can extract that into a page of its own to enable modular structure.
3. We can then add a capability to edit the posts.

Let us now start adding pages and create navigation.

We now have 2 sections - one to create posts and one to list the posts. We will have these 2 sections as different pages.

Create a file at the root level (inside **src/app** folder) and name it **app-routing.module.ts**

We can name it anything we like, but it is a convention and best practice to name it this way.



1. Create a class called **AppRoutingModule** and decorate it with **@NgModule** as this is essentially an angular module. Angular modules are basically building blocks which angular analyses to understand which features the app uses.

We can define our routes in the App Module, but its a good practice to not clutter App Module with too much code. So we will define route setup. Route configuration in a separate file and then import this in the App Module.

NgModule is imported from **@angular/core**.

import { NgModule } from '@angular/core';

@NgModule({

})

export class AppRoutingModule {}

1. Next, to enable routing feature import **RouterModule**.

import { RouterModule } from '@angular/router';

As the name suggests, this module helps us in setting up routing. It enables angular routing by maintaining route configurations.

## 2. Routes

Angular routes are not the same kind of routes that we see in the node/express app.

Angular routes might/ might not correspond to a backend service route.

Routes are JS objects where we defined for which URL which part of the app should be presented.

Define a new constant as below.

const routes: Routes = [

{ path: '', component: PostListComponent},

{ path: 'create', component: PostCreateComponent}

];

The constant that we define should be of type **Routes**. We also need to import Routes from **@angular/router**.

import { RouterModule, Routes } from '@angular/router';

Also import the referred components.

import { PostListComponent } from './posts/post-list/post-list.component';

import { PostCreateComponent } from './posts/post-create/post-create.component';

We now have 2 routes. The one with path as '' indicates the root of the application. If the domain for the application is [mydomain.com,](http://www.mydomain.com,) <http://mydomain.com> will load the PostListcomponent.

Now we will register this route object with angular router as shown below.

@NgModule({

imports: [RouterModule.forRoot(routes)]

})

For the app module to be able to import the routing module that we created, it needs to be exported too for it to be available outside of this routing modules.

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

Now in the app module (**app.module.ts**), we will import the App routing module.

In the imports section, add **AppRoutingModule**.

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

HttpClientModule,

BrowserAnimationsModule,

MatInputModule,MatCardModule,MatButtonModule,

MatToolbarModule, MatExpansionModule

]

VS IDE should automatically add the import for this module, but for some reason if it does not, we need to add the below import at the top of the file.

import { AppRoutingModule } from './app-routing.module';

Now we are all set to use our angular router with the routes that we defined.

To do that, we need to mark a place where angular should load the active routes. App component template would be the right place to do it as these routes are defined for the root.

Currently, we have the following selectors defined which are static.

<main>

<app-post-create></app-post-create>

<app-post-list></app-post-list>

</main>

We need to replace this and add a **hook** for the angular router to render its content. The hook is created with **router-outlet** directive provided by angular router package.

<main>

<router-outlet></router-outlet>

</main>

Now if we reload the app, the root of the app should load the Post list component.

If we add /create to the url, it should take us to the Post Create component.

But in the real app, this is not how we will navigate between pages. We will need to define links to load these components on clicking.

We can create links on the header.

In the **header.component.html**, make the following changes.

<mat-toolbar color="primary">

<span><a routerLink="">My Messages</a></span>

<span class="spacer"></span>

<ul>

<li>

<a mat-button routerLink="/create" routerLinkActive="mat-accent">New Post</a>

</li>

</ul>

</mat-toolbar>

Angular **routerLink** directive is provided by angular router. This converts a regular link to a angular link. Any click will be then interpreted by angular and the corresponding route will be activated.

## 3. Styling links

Let us create a style file for header component. In the header folder, create a file named **header.component.css** and add the following styles.

a {

text-decoration: none;

color: white;

}

ul {

list-style: none;

margin: 0;

padding: 0;

}

.spacer {

flex: 1 1 auto;

}

In the **header.component.ts**, link the newly created css file.

@Component({

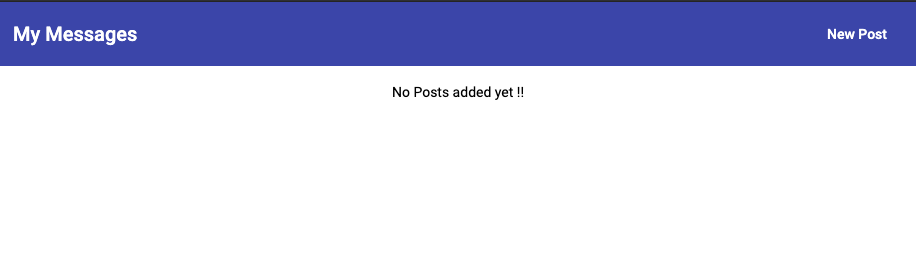
selector: 'app-header',

templateUrl: './header.component.html',

styleUrls: ['./header.component.css']

})

Now we should have nice looking links in the header which will allow us to switch between the 2 components. The page should look something like below.



## 3. Client side vs server side routing

We have routes created in the angular module. We also have routes defined in the backend (**app.js**) . These routes are not connected. They do different things.

Angular router parses the url for the app and render different things on the screen through JS. It does not render different pages, but the same page with different components. The routes defined in angular router is only known within the angular app context.

Server/ backend does not know of these routes. Server side routes are all about reading incoming requests and doing different things.

If for some reason we were to host both the angular app and the node backend to the same server, we should not use the same route name in frontend and backend app which conflicts with each other. Because server would not pass the same-named route to angular for processing.

## 4. Creating the edit form

We are now able to add new posts through post create component and then view it in Post list component. Now let us add functionality to make the posts editable.

The edit button is not functional at this moment. The expectation is that when we click edit, it should take us to the create post page but with pre-loaded post information for the post we want to edit. So essentially, it is a different route.

We will add a new route to the route object in **app-routing.module.ts** as below.

{ path: 'edit/:postId', component: PostCreateComponent}

Anything which would be a dynamic segment in the path is indicated to the angular router using a : (colon).

Since we are reusing the Post Create component, we should have a way to tell whether we are creating a new post or editing an existing post. Angular router provides us a way to identify that.

**ActivatedRoute**is an interface which contains the information about a route associated with a component loaded into an outlet and it can also be used to traverse the router state tree.

In Post Create component we can inject **ActivatedRoute** which will provide us information on the current route.

constructor(public postsService: PostsService,

public route: ActivatedRoute) {}

Activated route is imported from **@angular/router** package.

import { ActivatedRoute } from '@angular/router';

We want to know the current route information when the component is loaded so that we can render the component relevant to the current route. We can do that inside the constructor, but it is a good practice not to use the constructor for this.

Angular provides the **ngOnInit** method for the component to perform component load activities. ngOnInit is part of the interface **OnInit** which is part of **@angular/core**.

Change the class definition as below.

export class PostCreateComponent implements OnInit{

VS code should automatically add the import, but if it does not, add **OnInit** to Import statement from **@angular/core**

import { Component, OnInit } from '@angular/core';

We will add a private property called mode and default it to create.

private mode = 'create';

We will add another property to hold the postId.

private postId: string;

We will also add a property of type Post

post: Post;

Post will be imported from Post model.

import { Post } from '../post.model';

The component class code will look like below.

export class PostCreateComponent implements OnInit{

enteredTitle = '';

enteredContent = '';

// @Output() postCreated = new EventEmitter<Post>();

private mode = 'create';

private postId : string;

post: Post;

constructor(public postsService: PostsService,

public route: ActivatedRoute) {}

Now Implement ngOnInit as below.

ngOnInit(): void {

this.route.paramMap.subscribe((paramMap: ParamMap) => {

if(paramMap.has('postId')) {

this.mode = 'edit';

this.postId = paramMap.get('postId');

this.post = this.postsService.getPost(this.postId);

}

else {

this.mode = 'create';

this.postId = null;

}

});

}

paramMap is an observable and so we need to subscribe for it. It is an observable because the parameter on the URL can change while we are on the page.

Next, if we land into edit mode, we need to fetch the information of the post that we are trying to edit. We can get this information from post service. Where we will make a call to the backend to get the post that we are trying to edit. Ideally, we should try to get information within the frontend app and avoid any trips to backend while we can.

As of now, we do not have a method to fetch a single post. So, we will create a new method to fetch a singe post.

getPost(id: string) {

return {...this.posts.find(p => p.id === id)};

}

In post list component, change the edit button to a edit link as below. We will use a property binding for router link and assign the link.

<a mat-button color="primary" [routerLink]="['/edit/', post.id]">EDIT</a>

Now we are getting the posts and we have to pre-populate the form with the values from the post.

Since we are using the template driven approach, we have to do one-way binding on the **ngmodel**.

In **post-create.component.html**, change the **ngModel** binding for the post title as

[ngModel]="post?.title"

change the **ngModel** binding for the post content as

[ngModel]="post?.content"

Now if we go back to the app and try clicking the edit button, post create component should load in edit mode with pre-populated post info.

On submitting the post from the edit form, we will need to update the same post and not add a new post to the database.

So we will need a new backend route to update the posts.

In the **app.js** file of the backend, add the following function

app.put('/api/posts/:id', (req, res, next) => {

const post = new Post({

title: req.body.title,

content: req.body.content

});

Post.updateOne({\_id: req.params.id}, post)

.then(updatedPost => {

res.status(201).json({

message: 'Post Updated!!',

postId: updatedPost.\_id

});

});

});

On the frontend, we need a corresponding route for updating the post.

In **post.service.ts**, add a new function to update the post.

updatePost(id: string, title: string, content: string) {

const post: Post = {id: id, title: title, content: content};

this.http.put<{message: string, postId: string}>('http://localhost:3000/api/posts/' + id, post)

.subscribe(responseData => {

console.log(responseData);

const updatedPosts = [...this.posts];

const oldPostIndex = updatedPosts.findIndex(p => p.id == post.id);

updatedPosts[oldPostIndex] = post;

this.posts = updatedPosts;

this.postsUpdated.next([...this.posts]);

});

}

Also, in **post create component**, we will now add another action for updating the post.

We will now rename the **onAddPost** action to **onSavePost** (in both  **post.create.component.ts** and **post create component.html files).**

onSavePost(form: NgForm){

if(form.invalid) return;

if(this.mode === 'create') {

this.postsService.addPost(form.value.title, form.value.content);

}

else {

this.postsService.updatePost(this.postId, form.value.title, form.value.content);

}

form.resetForm();

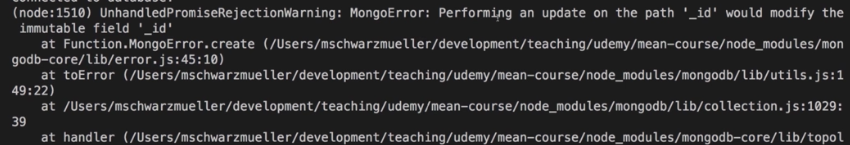
}

Now if we try editing a post and submit, we might see a error in the console. This is a CORS errors as we have not enabled PUT yet in the backend.

In **app.js** file of the backend, we will allow the PUT request by making the following update.

res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, PATCH, DELETE, OPTIONS');

Now if we try editing again, we might find an error on the backend server logs, something like below.



This is because in the put action in the app.js, we are creating a post from the incoming params, but not accounting for the \_id. \_id is a mongo specific identifier for a document, and it cannot be modified once created.

So let us use the same \_id for the original post and use it to update the post.

const post = new Post({

\_id: req.body.id,

title: req.body.title,

content: req.body.content

});

Now if we try editing, we should be able to see that the submit would successfully execute and we should be able to see the updated post.

## 5. Updating posts on the server

While we are on the edit page if we reload the page, we might lose the pre-populated post information. That is because, if we reload all our state is lost and the app refreshes.

And the latest version of the posts are only called in the post list component.

We now need a method on the server, which can give us the latest post information.

In **app.js** file of the server, add a new method to get post by id.

app.get('/api/posts/:id', (req, res, next) => {

Post.findById(req.params.id)

.then(post => {

if(post){

res.status(200).json(post);

}

else{

res.status(404).json({

message: 'post not found'

});

}

});

});

Now that we have a method on the backend, we will now call it from the frontend. Update the getPost method in **posts.service.ts** file to fetch from server rather than a local copy.

getPost(id: string) {

return this.http.get<{ \_id: string, title: string, content: string }>('http://localhost:3000/api/posts/' + id);

}

The above operation is asynchronous and we cannot return the post within the subscribe block as we cannot update the edit form once it loads.

So, we will not subscribe here, but return a observable which then we will subscribe to in the Post create component.

Now in **post.create.component**, make the following changes.

In ngOnInit, we will change the beelow line

this.Post = this.postsService.getPost(this.postId)

to

this.postsService.getPost(this.postId).subscribe(postData => {

this.isLoading = false

this.post = {id: postData.\_id, title: postData.title, content: postData.content }

});

Now we should be able to re-load edit page and see the expected behaviour.

## 6. Re-organizing backend route

As of now, in the backend, all the routes are defined in a single file - **app.js**. We can reorganise the folder structure and logically define route files.

Let us create a new folder called **routes** in the **backend** folder. Also create a file named **posts.js** inside the routes folder.

We will use a feature provided by express called router. Add the below lines in the file.

const express = require('express');

const Post = require('../models/post');

const router = express.Router();

Now move all the routes from **app.js**, to **posts.js**

Change all the concurrence of **app** to **router.**

Now export the router as below.

module.exports = router;

The file content will finally look like below.

const express = require('express');

const Post = require('../models/post');

const router = express.Router();

router.post('/api/posts', (req, res, next) => {

const post = new Post({

title: req.body.title,

content: req.body.content

});

post.save()

.then(createdPost => {

res.status(201).json({

message: 'Post Added !!',

postId: createdPost.\_id

});

});

});

router.get('/api/posts', (req, res, next) => {

Post.find()

.then(documents => {

res.status(200).json({

message: 'post sent successfully',

posts: documents

});

});

});

router.get('/api/posts/:id', (req, res, next) => {

Post.findById(req.params.id)

.then(post => {

if(post){

res.status(200).json(post);

}

else{

res.status(404).json({

message: 'post not found'

});

}

});

});

router.delete('/api/posts/:id', (req, res, next) => {

Post.deleteOne({\_id: req.params.id})

.then(() => {

res.status(200).json({

message: "Post Deleted!!"

});

});

});

router.put('/api/posts/:id', (req, res, next) => {

const post = new Post({

\_id: req.body.id,

title: req.body.title,

content: req.body.content

});

Post.updateOne({\_id: req.params.id}, post)

.then(updatedPost => {

res.status(201).json({

message: 'Post Updated !!',

postId: updatedPost.\_id

});

});

});

module.exports = router;

This will be then imported in app.js.

const postRoutes = require('./routes/posts');

At the bottom of the file, we will make express aware of the post routes.

app.use(postRoutes);

Also, we can fine tune it further. Since most of the routes have **/api/posts** as a common factor, we can make the following updates.

In **app.js**, add the **/api/posts** as a first argument while registering postRoutes.

app.use('/api/posts', postRoutes);

In **posts.js**, modify all the routes to strip off **/api/posts**.

router.post('', (req, res, next) => {

router.put('/:id', (req, res, next) => {

router.get('', (req, res, next) => {

router.get('/:id', (req, res, next) => {

router.delete('/:id', (req, res, next) => {

App should still be working as expected.

## 7. Adding Loading spinners

We need to show a progress indicator when creating, editing or loading posts.

Also, when we add a new post or edit a post, currently we stay in the same page. But it would be a good user experience to redirect the user to post list page.

Inject angular router into **post.service.ts**

constructor(private http: HttpClient, private router: Router) {}

VSCode should automatically add the import in the file.

import { Router } from '@angular/router';

Add the navigate method in the subscribe block of **addPost** and **updatePost** actions.

this.router.navigate(['/']);

Now to add a progress spinner, we will use Progress spinner from Material UI.

In **app.module.ts**, in the imports section, import **MatProgressSpinnerModule** which is imported from **@angular/material/progress-spinner**

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

HttpClientModule,

BrowserAnimationsModule,

MatInputModule,MatCardModule,MatButtonModule, MatToolbarModule,

MatExpansionModule, MatProgressSpinnerModule

],

**MatProgressSpinnerModule** import will be automatically added by VSCode. If not, manually add it at the top of the file among other import statements.

import { MatProgressSpinnerModule } from '@angular/material/progress-spinner'

Now that it is unlocked, we can now use it in our components.

To indicate the app when the posts are in loading mode and when posts have finished loading, we will add a new property.

In **post.create.component.ts** and **post.list.component.ts**, add the below property. Initial value will be false.

isLoading = false;

We will make updates to **ngOnit** action on both components by setting isLoading to true when it begin an operation and set it to false when it has completed the operation.

In **post-create.component.ts** make changes to both **ngOninit** and **onSavePost**

ngOnInit(): void {

this.route.paramMap.subscribe((paramMap: ParamMap) => {

if(paramMap.has('postId')) {

this.mode = 'edit';

this.postId = paramMap.get('postId');

this.isLoading = true;

this.postsService.getPost(this.postId).subscribe(postData => {

this.isLoading = false

this.post = {id: postData.\_id, title: postData.title, content: postData.content }

});

}

else {

this.mode = 'create';

this.postId = null;

}

});

}

onSavePost(form: NgForm){

if(form.invalid) return;

this.isLoading = true;

if(this.mode === 'create') {

this.postsService.addPost(form.value.title, form.value.content);

}

else {

this.postsService.updatePost(this.postId, form.value.title, form.value.content);

}

form.resetForm();

}

In **post-list.component.ts**, make changes to **ngOninit** as below

ngOnInit(){

this.isLoading = true;

this.postsService.getPosts();

this.postsSub = this.postsService.getPostUpdateListener()

.subscribe((posts: Post[]) => {

setTimeout(()=>{ this.isLoading = false }, 2000);

this.posts = posts;

});

}

In **post-create.component.html**, add **mat-spinner** and load it only when **isLoading** is true and load the form field when **isLoading** is false.

<mat-card>

<mat-spinner \*ngIf="isLoading"></mat-spinner>

<form (submit)="onSavePost(postForm)" #postForm="ngForm" \*ngIf="!isLoading">

<mat-form-field>

<input matInput type="text" name="title" [ngModel]="post?.title"

placeholder="Post Title"

required minlength="3" #title="ngModel">

<mat-error \*ngIf="title.invalid">Enter a title</mat-error>

</mat-form-field>

<mat-form-field>

<textarea matInput rows="4" name="content" [ngModel]="post?.content"

placeholder="Post content"

required #content="ngModel"></textarea>

<mat-error \*ngIf="content.invalid">Please enter content</mat-error>

</mat-form-field>

<button mat-raised-button

color="primary"

type="submit">

Save Post

</button>

</form>

</mat-card>

In css files for the component, we can center the spinner

mat-spinner {

margin: auto;

}

Now in **post-list.component.html,** make similar changes

<mat-spinner \*ngIf="isLoading"></mat-spinner>

<mat-accordion multi="true" \*ngIf="posts.length > 0 && !isLoading">

<mat-expansion-panel \*ngFor="let post of posts">

<mat-expansion-panel-header>

{{ post.title }}

</mat-expansion-panel-header>

<p>{{ post.content }}</p>

<mat-action-row>

<a mat-button color="primary" [routerLink]="['/edit/', post.id]">EDIT</a>

<button mat-button color="warn" (click)="onDelete(post.id)">DELETE</button>

</mat-action-row>

</mat-expansion-panel>

</mat-accordion>

<p class="info-text mat-body-1" \*ngIf="posts.length <= 0 && !isLoading"> No Posts added yet !!</p>

And in **post-list.component.css**

mat-spinner {

margin: auto;

}

With these changes, now we will see the spinner displaying while creating, editing and listing posts.

# Appendix

Based on your node and npm version, you might find an error as shown below when trying to build/run the app.

To fix it, make the highlighted changes in **tsconfig.json** file.

# 

