*Adding User Authentication*

[Adding User Authentication 2](#_Toc2135393760)

[Adding Signup and Signin routes 2](#_Toc816215656)

[Creating the login form 3](#_Toc10993356)

[Handling the User inputs 5](#_Toc81740650)

[Creating the Signup form 6](#_Toc1573054117)

[Creating the user model 7](#_Toc649366202)

[Signup Route 8](#_Toc385862960)

[Connecting Angular to the backend 10](#_Toc1945638427)

[Understanding SPA authentication 12](#_Toc611549720)

[Implementing SPA Authentication 14](#_Toc459409498)

[Adding middleware to protect routes 16](#_Toc1091721921)

[Adding the token to the requests from Front end 17](#_Toc470046279)

[UI updates to reflect authentication 20](#_Toc1622957487)

[Connecting logout button to auth status 25](#_Toc1856683462)

[Adding route guards and redirection 25](#_Toc194071277)

[Redirection 25](#_Toc1875776393)

[Route Guards 26](#_Toc1113899191)

[Reflecting token expiration in the UI 27](#_Toc1688752238)

[Storing the token in Local storage 28](#_Toc1714985314)

# Adding User Authentication

* Till now, we have built a nice-looking social networking app. Now we will implement Application security by implementing User Authentication.
* As of now, we do not have users defined for the system and anyone can create a post, edit a post or delete the post.
* We need to implement a mechanism where only the creator of the post should be able to edit/delete the post.

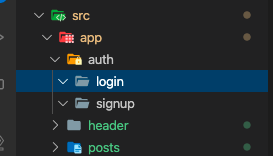
Let us start by adding an authentication form for signup and login and thereafter we will see how we can create users, how we secure our app how we authorize certain actions like creating and deleting post and so on.

## Adding Signup and Signin routes

First, let us add 2 new links and corresponding routes in our angular app for SignUp and SignIn. We shall place these links next to the **New Post** link on the header.

In our angular code, let us add a folder in the root called “**auth”** where we will place all our code for authentication.

1. Create 2 sub-folders under auth folder  
    a) signup - which will hold signup component.  
    b) login - which will hold login component.



1. Let us first create the login component. Create 3 files under login folder.  
    **login.component.html  
    login.component.ts**  
    **login.component.css**

On the typescript file, we will create a component called LoginComponent and decorate it with @Component to make it a component, just like how we were doing it in all the previous sessions.

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

@Component({

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

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

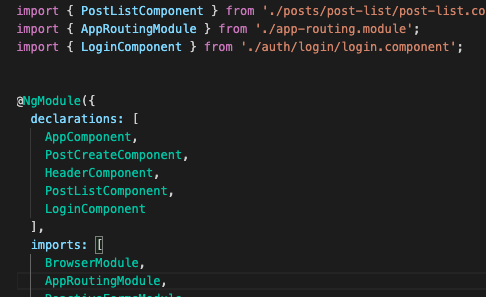
})

export class LoginComponent {

}

We will use routing to load this component, so we will not define a selector for the login component like we did with the other components.

Now we need to register this component in the App Module. Let us register this module in the declaration section in **app.module.ts.**



Now that we have registered this module, let us now create a template and loading it via routing.

## Creating the login form

Here we will create a form using a template driven approach

(Copy the form from post create component html and make following modifications.  
1. Keep the mat-card and mat-spinner

1. Remove **submit**, **[formgroup]=”form”, onSavePost()** from **form** tag.
2. Change the type of input to email, and remove the **formControlName** attribute.
3. Remove the condition and change the associated error message to

<mat-error \*ngIf="">Enter a valid email</mat-error>

1. Remove the button, input and image preview controls for the image upload.
2. For the next formfield change the textarea control to a input control of password type.

<input matInput type="password" placeholder="Enter the Password">

<mat-error \*ngIf="">Please enter a valid password</mat-error>

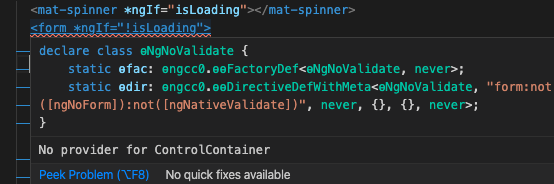
1. Change the submit button text to **Login**.
2. In the Typescript file, add a isLoading flag to remove the error from the template.

export class LoginComponent {

isLoading = false;

}

You would see the error below in the template.



This is because of the way we have defined the template to be a Template driven from approach and hence it needs to import **FormsModule** in app module to enabled Template driven approach.

In app.module.ts, make the following changes.

1. Import **FormsModule** from @angular/Forms
2. Add it to the imports list.

import { ReactiveFormsModule, FormsModule } from '@angular/forms';

imports: [

BrowserModule,

AppRoutingModule,

ReactiveFormsModule, FormsModule,

The error from the template should now be gone.

Now we would enable routing for this component. In the app-routing.module.ts, add a new path for login.

{ path:'login', component: LoginComponent}

Now we should be able to load the component by typing **/login**

Now, in the header template, we would add a Login link.

<li>

<a mat-button routerLink="/login" routerLinkActive="mat-accent">Login</a>

</li>

We can see a functional login link now and on clicking displays the Login form . We would need to style it better.

In the header css, for the **ul** style, add **display: flex**

ul {

list-style: none;

margin: 0;

padding: 0;

display: flex;

}

Also, in the **login.component.css**, add the following styles.

mat-form-field {

width: 100%;

}

mat-spinner {

margin: auto;

}

Now the form should look styled as desired.



## Handling the User inputs

Now that we have our login form ready, we need to implement logic to handle user input.

1. Create a submit event for the form and bind an event listener.
2. Create a local reference for the form and pass it to the event that is created.

<form (submit)="onLogin(loginForm)" #loginForm="ngForm" \*ngIf="!isLoading">

1. Create local reference for the input fields.   
   For this to work, we also need to add **ngModel** and **name** attributes to the input field to register this as an input.   
   Also add **required** and **email** validators to the input field.

It will finally look like below.

<input matInput type="text" ngModel name="email"

placeholder="Enter the User ID" #emailInput="ngModel"

required email>

Now the local reference for the input can be used as the condition for the inline error message.

<mat-error \*ngIf="emailInput.invalid">Enter a valid email</mat-error>

1. Repeat the same for password field.

<input matInput type="password" name="password" ngModel

placeholder="Enter the Password" #passwordInput="ngModel"

required>

<mat-error \*ngIf="passwordInput.invalid">Please enter a valid password</mat-error>

Let us now verify if the validators work as expected.

Once we confirm, we can now define the onLogin action in the ts file and console log the form value to check if we are able to access the user input.

onLogin(form: NgForm){

console.log(form.value);

}

Now that we have everything working as expected, we can move on and create the signup form.

## Creating the Signup form

Signup screen will look a lot like the login screen that we created.

1. So we will now copy all 3 files from the login component and use it to create the signup screen and rename them accordingly.  
     
   A) Rename the files

B) Rename the submit action and button name in html.

<form (submit)="onSignup(signupForm)" #signupForm="ngForm" \*ngIf="!isLoading">

and

<button mat-raised-button color="primary" type="submit" \*ngIf="!isLoading">

Sign Up

</button>

1. Register the signup component in app module by adding it in the declarations section.

import { SignupComponent } from './auth/signup/signup.component';

1. Create a route for the signup component.

{ path:'signup', component: SignupComponent}

1. Now we will create a link for signup in the header template.

<li \*ngIf="!userIsAuthenticated">

<a mat-button routerLink="/login" routerLinkActive="mat-accent">Login</a>

</li>

<li \*ngIf="!userIsAuthenticated">

<a mat-button routerLink="/signup" routerLinkActive="mat-accent">Sign Up</a>

</li>

Our app should now have links for the signup and login with the component loading when we click on these links.

A point to note here, as we get comfortable with development, we can optimize this code by creating a shared component (named form or something) and refer them in both login and signup screen so that we can reduce the code as per the DRY principle.

Now that the front end is configured, its time to create corresponding backend routes, handle users and store user info in database.

## Adding the user model

We added our frontend code for signup and login. Now we want to create routes for creating users and login.

In the nodejs app, just like we have **post.js** file which contains routes to work with posts, we will now create another file named **user.js** which will contain routes to work with user data and authentication.

In the user.js, we will copy the below 3 lines from post.js.

const express = require('express');

const router = express.Router();

module.exports = router;

We will now add 2 routes, 1 for signup and 1 for login. Let our routes structure be as below.

**api/user/login**

**api/user/signup**

In app.js, just like the way we configured post routes, we will configure the user routes.

At the top of the page, define a constant as below.

const userRoutes = require('./routes/user');

And at the bottom of the page, register the user route as below.

app.use('/api/users', userRoutes);

Any request where the path starts with **api/users** will be directed to userRoutes.

**Next** in the **user.js** create 2 post routes like below.

router.post("/signup", (req, res, next) => {

});

router.post("/login", (req, res, next) => {

});

### Signup Route

What do we do in the signup function? We will create new users and store in the database. For that we now need a new mongoose model.

In the models folders, add a new file - **user.js.** We can copy the code from the posts.js model file and name the model as **userSchema**.

const mongoose = require('mongoose');

const userSchema = mongoose.Schema({

email: { type: String, required: true, unique: true },

password: { type: String, required: true }

});

module.exports = mongoose.model('User', userSchema);

A point to note here is unlike the required validator which actually validates, “**unique”** does not validate the email to be unique. It does not automatically throw an error when an existing email id will be added. By informing mongoose that this field will be unique, Mongoose and mongoDB will use it for internal optimization and performance improvement.

However, we would indeed need to stop inserting a email which already exists. In order to implement that, let us use a third party package **mongoose-unique-validator**

npm i --save mongoose-unique-validator

Once it is installed, we can now use it to validate email field. In **user.js**, add this constant at the top of the page.

const uniqueValidator = require('mongoose-unique-validator');

Now we can use this as a plugin to the schema which is a mongoose feature. This would be nothing but an extra hook/functionality that runs on the schema before the data is inserted into the Database.

Now call the plugin on the schema with the validator that we have defined.

userSchema.plugin(uniqueValidator);

Later we will demonstrate what happens when we try to insert a duplicate email.

Now that we have user model created, let us work on inserting the user to DB when we receive the request for signup.

In **user.js** route, import the newly created user model.

const User = require('../models/user');

In the /signup route, let us capture user info from the incoming request.

const user = new User({

email: req.body.email,

password: req.body.password

});

If we look at the code above, there is one problem that we see in the way that we will use the incoming password. We are trying to use it as a plain text.

In order to avoid saving it in its raw form, we would need to encrypt before saving it.

We will now install a package called bcrypt which will provides us encryption functionalities which we can use in nodeJS.

Run the command - **npm i --save bcrypt**

Once installed we can import it on our user route.

const bcrypt = require('bcrypt');

Now we can update the signup route as below.

router.post("/signup", (req, res, next) => {

bcrypt.hash(req.body.password, 10)

.then(hash => {

const user = new User({

email: req.body.email,

password: hash

});

user.save()

.then(resp => {

res.status(201).json({

message:'User Created',

result: resp

})

})

.catch(err => {

res.status(500).json({

error: err

})

});

})

});

Now that we have made update to the backend, let us connect the frontend with the backend route.

## Connecting Angular to the backend

Now on the **auth** folder of the angular app, in the signup component we have the **onSignup** function. We need to call the newly created backend route from this function.

Since this is a new functionality for the app, we can now create a service on angular app to make sevice calls related to auth functionalities.

First, create a mode to hold the auth data. Inside auth folder, create a file **auth-data.model.ts** and add the following content.

export interface AuthData {

email: string,

password: string

}

Next, Create a service file in the auth folder - **auth.service.ts**

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

import { HttpClient } from '@angular/common/http';

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

import { AuthData } from './auth-data.model';

@Injectable({providedIn: 'root'})

export class AuthService {

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

createUser(email: string, password: string) {

const authData: AuthData = {

email: email,

password: password

};

this.http.post("http://localhost:3000/api/users/signup", authData)

.subscribe(response => {

console.log(response);

});

}

}

Now we need to connect the Auth service to the sign up component. On the signup component.ts, inject the newly created Auth service

Add the constructor and inject Auth Service.

constructor(public authService: AuthService) {}

Modify the onSignup method as below.

onSignup(form: NgForm){

if (form.invalid){

return;

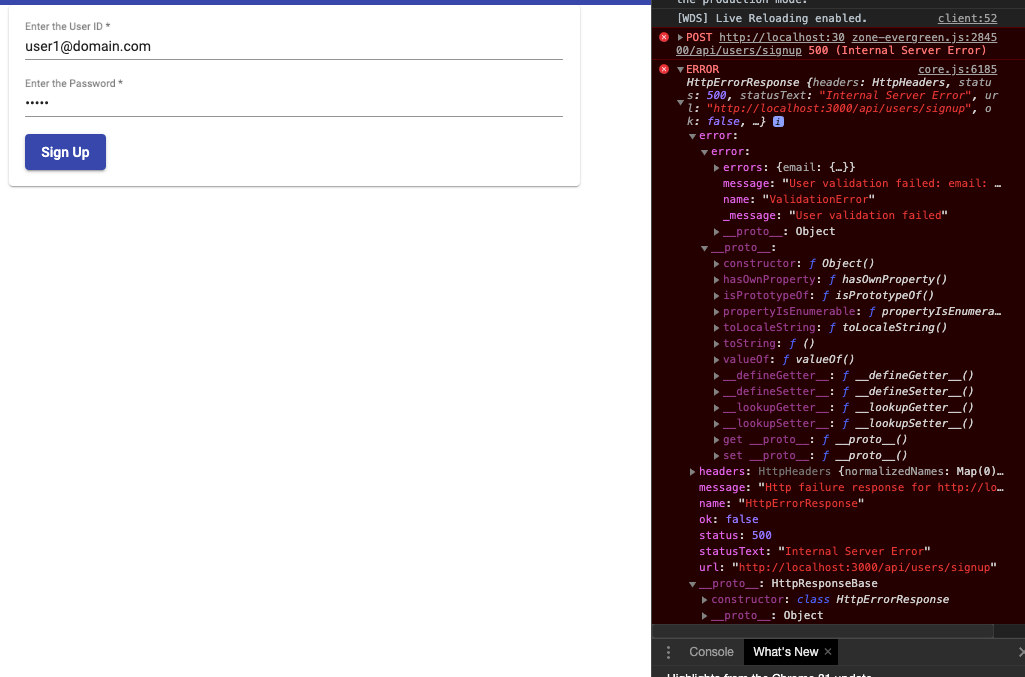
}

this.authService.createUser(form.value.email, form.value.password);

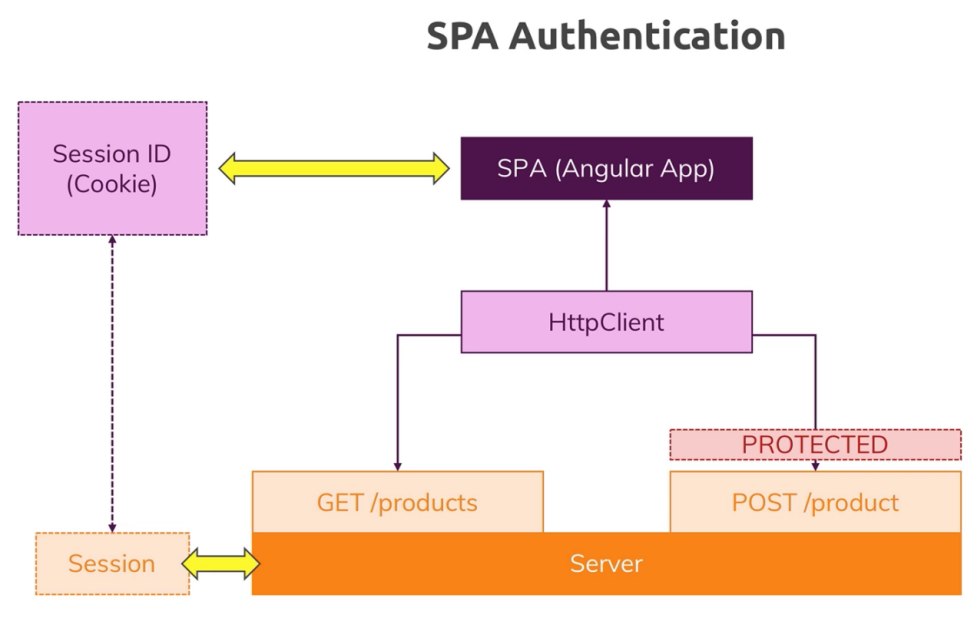
}

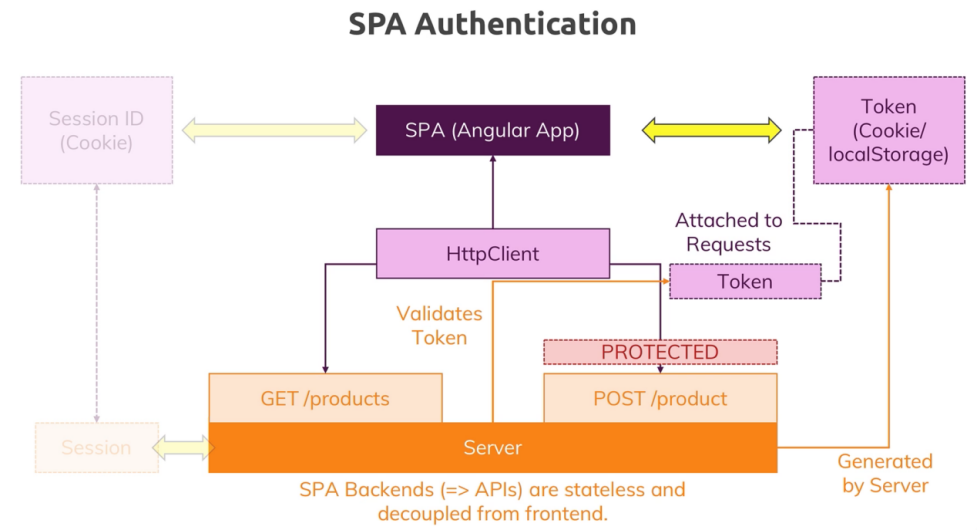
Now if we try to create a user on signup page, we should be able to see the console log for the success response object.

Now if we try to create a user with the same email, the opeartion should fail and console should display user validation error.



## Understanding SPA authentication





### Implementing SPA Authentication

The goal here is to create a token in the backend and return to the user on successful login. Let us now create a route for login action.

If you wish to learn more about JWT, please visit <https://jwt.io/>

To help us with creating a JWT, we can install a package in our node app.

npm i --save jsonwebtoken

Add the line below in **user.js** require section.

const jwt = require('jsonwebtoken');

The login route would look like below.

router.post("/login", (req, res, next) => {

let fetchedUser;

//Validate the incoming credentials with DB

User.findOne({ email: req.body.email})

.then(user => {

if(!user) {

return res.status(401).json({

message: "Authentication Failed!!!"

});

}

fetchedUser = user;

return bcrypt.compare(req.body.password, user.password)

})

.then(result => {

if(!result) {

return res.status(401).json({

message: "Authentication Failed!!!"

});

}

const token = jwt.sign({

email: fetchedUser.email,

userId: fetchedUser.\_id

},

'test\_secret\_key',

{

expiresIn: '1h'

}

);

res.status(200).json({

token: token

});

})

.catch(err => {

return res.status(401).json({

message: "Authentication Failed!!!"

});

});

});

Now that the backend is configured to enable login, we will now create login action on the front end.

In **auth.service.ts**, create the login action like below.

login(email: string, password: string) {

const authData: AuthData = {

email: email,

password: password

};

this.http.post("http://localhost:3000/api/users/login", authData)

.subscribe(response => {

console.log(response);

});

}

Now in the login folder, **login.component.ts** make the following changes.

* Inject the AuthService, like we did in the signup component.

constructor(public authService: AuthService) {}

* Call the login service action

onLogin(form: NgForm){

if (form.invalid){

return;

}

this.authService.login(form.value.email, form.value.password);

}

Now, if we try to login with a user id and password which we created earlier using the signup action, we should see in the console the result of the action.

## Adding middleware to protect routes

Now that the front end receives token from login action, all subsequent requests must start sending this token in the requests to the server. Only such requests will be treated as a valid request.

This protection will only be applied for selected routes. Auth routes (**signUp and login**) will remain open for the requests without the token, as these 2 routes when called will not contain a token yet.

So we will now add a middleware and use it across routes where we expect the user to be authenticated.

In the backend folder, add a new folder called **middleware** (we can name it anything we want), which will contain a function which will parse the request and determine if the request is allowed to proceed or should be rejected.

A typical middleware in nodejs will look like below.It is nothing but a function which executes on a incoming requests.

module.exports = (req, res, next) => {

}

We will add a file **check-auth.js** in the **middleware** folder and update it with the content below.

const jwt = require('jsonwebtoken');

module.exports = (req, res, next) => {

try {

//a tokec typically would look like "Bearer aduiuskfoab792"

const token = req.headers.authorization.split(" ")[1];

jwt.verify(token, 'test\_secret\_key');

next();

} catch (error) {

res.status(401).json({

message: "Token Invalid!!"

});

}

}

Now that we have the validation middleware created, we can now use it in the post routes.

In **posts.js file** in the backend, import the **check-auth** middleware.

const checkAuth = require('../middleware/check-auth');

Now you can apply the reference checkAuth to all the routes which needs to have a authenticated request, like shown below.

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

We will add it to the post, put and the delete route and allow unauthenticated users to only see the posts.

router.post('', checkAuth, multer({storage: storage}).single("image"), (req, res, next) => {

router.put('/:id', checkAuth, multer({storage: storage}).single("image"),

Now in the app, when we try to add a new post, we should see authentication error on browser console.

## Adding the token to the requests from Front end

Now we will start passing the token received from the login to the requests made to the backend.

On the frontend app, we will add a private variable in the **auth.service.ts** to store the token.

private token: string;

And then update the login function as below.

login(email: string, password: string) {

const authData: AuthData = {

email: email,

password: password

};

this.http.post<{token: string}>("http://localhost:3000/api/users/login", authData)

.subscribe(response => {

console.log(response);

const token = response.token;

this.token = token;

});

}

Also, we will add a function to expose the token to other parts of the app.

getToken() {

return this.token;

}

Now we will extract the token in the post service (**posts.service.ts**).

We can now get this token in the **posts.service.ts** by injecting auth service and adding the extracted token in the headers of each of the outgoing requests.

Alternatively, we can do it with an angular feature called **Interceptor** for the Http client.

Interceptors are functions which run before executing a http request.

We will add a file in the auth folder - **auth-interceptor.ts** and add the content below.

import { HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http';

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

import { AuthService } from './auth.service';

@Injectable()

export class AuthInterceptor implements HttpInterceptor{

constructor(private authService: AuthService) {}

intercept(req: HttpRequest<any>, next: HttpHandler) {

const authToken = this.authService.getToken();

const authRequest = req.clone({

headers: req.headers.set('Authorization', 'Bearer ' + authToken)

});

return next.handle(authRequest);

}

}

Interceptors are just like a middlewares that we saw in nodejs, except that it works on an outgoing request instead of incoming request.

Now we have to inject/provide/register this interceptor as a service.

We do it by adding this **app.module.ts**. In the provider section, change as below.

providers: [{

provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptor, multi: true

}],

HTTP\_INTERCEPTORS are imported from **@angular/common/http**

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

**multi: true** tell angular to use existing interceptors in addition to the custom interceptors. Now all the requests would receive the token.

If we test the angular app now, console will show a CORS error stating that the header is invalid. We need to now add the “Authorization” header in the allowed list.

In nodeJS,in app.js, allow the Authorization header to the **Access-Control-Allow-Headers** policy.

res.setHeader('Access-Control-Allow-Headers',

'Origin, X-Requested-With, Content-Type, Accept, Authorization');

Now we should be able to create, edit or delete posts. Also, if we examine the header of the outgoing requests in browser developer tools, we can see the Auth header getting passed with the token that we received.

## UI updates to reflect authentication

At this point, UI does not reflect the authentication status. A user cannot tell what he is allowed to do and what he is not. We need to hide **login** and **signup** links once we are logged in, and display the New Post link only when we are authenticated.

Also the token that we generated is stored in the memory as it is stored by the auth service. If we refresh the page, we lose the token. So we need to store the token somewhere else for the app to account for page refresh.

We will now make the following changes

1. Hide **Login** and **Signup** after login.
2. Add a **Logout** button which would delete the token.
3. Store the token somewhere.

In **header.component.html**, add a new button for logout.

<li>

<button mat-button>Sign Up</button>

</li>

In **header.component.ts**, we will inject auth service.

constructor(private authService: AuthService) {}

Just like how we listen/subscribe for post updates, we also need to look for token updates as to keep the authentication status up-to-date at all times.

In the **auth.service.ts**, we will add a listener to listen for token updates.

private authStatusListener = new Subject<boolean>();

**Subject** is imported from **rxjs** package. Add this import in **auth.service.ts,** if VSCode has not already added it.

import { Subject } from 'rxjs';

We will add a method to return the auth status to help other components to retrieve the auth status.

getAuthStatusListener() {

return this.authStatusListener.asObservable();

}

In the login function, we will now emit the authentication status for the listeners.

this.authStatusListener.next(true);

Login function will now look like -

this.http.post<{token: string}>("http://localhost:3000/api/users/login", authData)

.subscribe(response => {

console.log(response);

const token = response.token;

this.token = token;

this.authStatusListener.next(true);

});

Now in the **header.component.ts**, make the following changes.

Implement OnInit and OnDestroy interfaces.

export class HeaderComponent implements OnInit, OnDestroy{

userIsAuthenticated = false;

private authListernerSubs: Subscription;

constructor(private authService: AuthService) {}

ngOnInit(): void {

this.authListernerSubs = this.authService.getAuthStatusListener()

.subscribe(isAutheticated => {

this.userIsAuthenticated = isAutheticated;

})

}

ngOnDestroy(): void {

this.authListernerSubs.unsubscribe();

}

}

Subscription is imported from rxjs package. Add this import in **header.component.ts,** if VSCode has not already added it.

import { Subscription } from 'rxjs';

In the header.component.html, use the **userIsAuthenticated** property to display links appropriately.

<ul>

<li \*ngIf="userIsAuthenticated">

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

</li>

<li \*ngIf="!userIsAuthenticated">

<a mat-button routerLink="/login" routerLinkActive="mat-accent">Login</a>

</li>

<li \*ngIf="!userIsAuthenticated">

<a mat-button routerLink="/signup" routerLinkActive="mat-accent">Sign Up</a>

</li>

<li \*ngIf="userIsAuthenticated">

<button mat-button>Logout</button>

</li>

</ul>

After the app reloads, now in the header we should see only signup and login button. When we login, we will see onlt New Post and logout buttons.

Also, in the **postlist.component.ts** we can implement the same logic, to hide the edit and delete option for the post unless the user is logged in.

In **postlist.component.ts, i**nject auth service as shown below.

constructor(public postsService: PostsService, private authService: AuthService) {

}

Auth service need to be imported in the file. Add this import in the file**,** if VSCode has not already added it.

import { AuthService } from 'src/app/auth/auth.service';

Declare a private property for the subscription and a variable to hold authentication status.

private authStatusSub: Subscription;

userIsAuthenticated = false;

In **ngOnInit** action, add the subscription after the posts subscription snippet.

ngOnInit(){

this.isLoading = true;

this.postsService.getPosts(this.postsPerPage, 1);

this.postsSub = this.postsService.getPostUpdateListener()

.subscribe((postData: {posts: Post[], postCount: number}) => {

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

this.posts = postData.posts;

this.totalPosts= postData.postCount;

});

this.userIsAuthenticated = this.authService.getIsAuth();

this.authStatusSub = this.authService.getAuthStatusListener()

.subscribe(isAuthenticated => {

this.userIsAuthenticated = isAuthenticated;

});

}

In **ngOnDestroy**, unsubscribe to the subscription added above.

ngOnDestroy() {

this.postsSub.unsubscribe();

this.authStatusSub.unsubscribe();

}

Now we can use the userIsAuthentiated property in **post.list.component.html** to use the authentication status.

<mat-action-row \*ngIf="userIsAuthenticated">

<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>

Now if we reload the app and are not logged in, we will now see the posts without the edit and delete button.

You might also see that if we login now and check, we might still see edit and delete buttons not showing up. This is due to the post list component is subscribing for auth status on initialization only.

To fix this, In **auth.service.ts**, make the following changes.

Declare a private property for auth status.

private isAuthenticated = false;

Add a function to return the **isAuthenticated** status.

getIsAuth() {

return this.isAuthenticated;

}

And in the login action, make the following changes to set the auth status only when there is a valid token.

if(token){

this.authStatusListener.next(true);

this.isAuthenticated = true;

}

Login action will look like below.

login(email: string, password: string) {

const authData: AuthData = {

email: email,

password: password

};

this.http.post<{token: string}>("http://localhost:3000/api/users/login", authData)

.subscribe(response => {

console.log(response);

const token = response.token;

this.token = token;

if(token){

this.isAuthenticated = true;

this.authStatusListener.next(true);

}

});

}

Now in post-list.component.ts, we can look for auth status by calling the new action.

this.userIsAuthenticated = this.authService.getIsAuth();

Add this line in the ngOnInit action as shown below.

ngOnInit(){

this.isLoading = true;

this.postsService.getPosts(this.postsPerPage, 1);

this.postsSub = this.postsService.getPostUpdateListener()

.subscribe((postData: {posts: Post[], postCount: number}) => {

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

this.posts = postData.posts;

this.totalPosts= postData.postCount;

});

this.userIsAuthenticated = this.authService.getIsAuth();

this.authStatusSub = this.authService.getAuthStatusListener()

.subscribe(isAuthenticated => {

this.userIsAuthenticated = isAuthenticated;

});

}

Now if we re-login, we should see the edit and delete button showing up as expected.

## Connecting logout button to auth status

We will now add a logout button in the header to log the user out.

In the **header.component.html,** add **onLogut** click action.

<button mat-button (click)="onLogout()">Logout</button>

In **header.component.ts**, add the **onLogout** action, which will clear the token and also inform the change in the auth status to all the interested components.

onLogout() {

this.authService.logout();

}

Add a logout function in auth.service.ts.

logout() {

this.token = null;

this.isAuthenticated = false;

this.authStatusListener.next(false);

}

Now in the app, we can see that on click of logout we will see the desired behaviour.

## Adding route guards and redirection

We now need to implement redirections when user login or logout. Also, we need to stop users from loading the page by enetering the URLs directly or through bookmarks unless they are logged in.

### Redirection

Auth service is the best place to implement redirection as it contains both login and logout actions.

To implement redirection, let us start by injecting angular router in **auth.service.ts**

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

Router is imported from **@angular/router** package.

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

In login (inside the token check) and logout actions, add the navigation code.

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

Now we can see the login and logout actions redirecting the user to home page.

login(email: string, password: string) {

const authData: AuthData = {

email: email,

password: password

};

this.http.post<{token: string, expiresIn: number}>("http://localhost:3000/api/users/login", authData)

.subscribe(response => {

console.log(response);

const token = response.token;

this.token = token;

if(token){

const expiresInDuration = response.expiresIn;

this.setAuthTimer(expiresInDuration);

this.isAuthenticated = true;

this.authStatusListener.next(true);

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

}

});

}

logout() {

this.token = null;

this.isAuthenticated = false;

this.authStatusListener.next(false);

clearTimeout(this.tokenTimer);

this.clearAuthData();

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

}

### Route Guards

A guard behaves as a service. Angular provide interfaces that classes can implement which forces the classes to add some methods which angular router can execute to check if it can proceed or not.

Create file in the auth folder - **auth.guard.ts**.

import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router }

from '@angular/router';

import { Observable } from 'rxjs';

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

import { AuthService } from './auth.service';

@Injectable()

export class AuthGuard implements CanActivate{

constructor(private authService: AuthService, private router: Router) {

}

canActivate(

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot

): boolean | Observable<boolean> | Promise<boolean> {

const isAuth = this.authService.getIsAuth();

if(!isAuth) {

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

}

return isAuth;

}

}

If the value returned is true/observable or promise which will return true, the router can load the route which it guards.

In cases where it can return false, we should also provide an alternative redirection. Else the router will halt and not know what to do next.

Now we need to add this guard to the app routing module.

Update the @NgModule section in the **app-routing.module.ts**.

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule],

providers: [AuthGuard]

})

Now that Auth guard is provided for the whole app, we can now use it against the routes that we want to protect.

Update the create and edit routes in **app-routing.module.ts**. Once updated we can try to go to /create or /edit pages in browser, but should get redirected to login.

{ path: 'create', component: PostCreateComponent, canActivate: [AuthGuard]},

{ path: 'edit/:postId', component: PostCreateComponent, canActivate: [AuthGuard]},

## Reflecting token expiration in the UI

We have configured the token to expire in 1 hour. In the case where token expires, the authentication status should be updated accordingly.

We will now update the UI to account for the token expiration.

In the nodeJs app, **user.js** file in the **routes** folder , in addition to returning the token, we will also return the **expiresIn** property in seconds which will tell the how long the token is valid.

In the **login** route in **user.js**, update the token returning code as shown below.

res.status(200).json({

token: token,

expiresIn: 3600

});

In the UI in **auth,service.ts**, add a private property in this file to store the timer/duration elapsed.

private tokenTimer: any;

inside the login action token check **,** add the following lines. We need to add timer and call logout when the duration has elapsed.

setTimeout needs the duration to be supplied in milliseconds, hence multiply by 1000.

const expiresInDuration = response.expiresIn;

this.tokenTimer = setTimeout(() => {

this.logout();

}, expiresInDuration \* 1000)

And in the logout action, clear the timer when user clicks logout. Add the line below before navigation.

clearTimeout(this.tokenTimer);

Now, to prove that this works, we can reduce the timer by removing the multiplier (\* 1000) and making it 3.6 secs. Now when we login, we see that we get logged out in 3.6 seconds.

## Storing the token in Local storage

We still have a problem to solve - we lose the token if the page refreshes as the token in stored in the memory. To solve this, we can store the token in local storage of the browser.

Local storage is a storage managed by the browser and is made accessible through JS.

In **auth.service.ts**, add the below 2 actions to set and remove auth data.

private saveAuthData(token: string, expirationDate: Date) {

localStorage.setItem('token', token);

localStorage.setItem('expiration', expirationDate.toISOString());

}

private clearAuthData() {

localStorage.removeItem('token');

localStorage.removeItem('expiration');

}

private getAuthData() {

const token = localStorage.getItem('token');

const expiration = localStorage.getItem('expiration');

if(!token || !expiration) {

return;

}

return {

token: token,

expirationDate: new Date(expiration)

}

}

In the login action, add the following lines before navigation

const now = new Date();

const expirationDate = new Date(now.getTime() + expiresInDuration \* 1000);

this.saveAuthData(token, expirationDate)

Now when we login and check the Application Data in the browser storage and we will see that the auth data is stored.

We can implement the login to auto authenticate the user when the token is not expired.

Add the below 2 actions in auth.service.ts

autoAuthUser() {

const authInfo = this.getAuthData();

const now = new Date();

const isInFuture = authInfo.expirationDate > now;

if(isInFuture) {

this.token = authInfo.token;

this.isAuthenticated = true;

this.authStatusListener.next(true);

}

}

private setAuthTimer(duration: number) {

this.tokenTimer = setTimeout(() => {

this.logout();

}, duration \* 1000);

}

In login action, where we are setting the timer, we can replace it with calling the **setAuthTimer** function.

this.setAuthTimer(expiresInDuration);

Update autoAuthUser function as below

autoAuthUser() {

const authInfo = this.getAuthData();

if(!authInfo) return;

const now = new Date();

const expiresIn = authInfo.expirationDate.getTime() - now.getTime();

if(expiresIn > 0) {

this.token = authInfo.token;

this.isAuthenticated = true;

this.setAuthTimer(expiresIn/1000);

this.authStatusListener.next(true);

}

}

Now that we have the setup for auto authentication, we need a place to call it. The best place to call it is in the app component as this is the first module that loads and this is the best place to perform basic app initialization activities.

In the **app.module.ts**, make the following changes

* Inject Authservice
* Implement OnInit and call autoAuthUser action from authservice.

export class AppComponent implements OnInit{

constructor(private authService: AuthService) {}

ngOnInit(): void {

this.authService.autoAuthUser();

}

}

We can now see that once logged in, if we open the page in a new browser tab, user should be still logged in.

In **header.component.ts**, make the following update in ngOnInit for header links to update.

this.userIsAuthenticated = this.authService.getIsAuth();

Now we should see that the header accurately reflects auth status on auto authentication.

ngOnInit(): void {

this.userIsAuthenticated = this.authService.getIsAuth();

this.authListernerSubs = this.authService.getAuthStatusListener()

.subscribe(isAuthenticated => {

this.userIsAuthenticated = isAuthenticated;

})

}

Now that we have explored aspects of user authentication features, let us move on and explore authorization.