

Table of Contents

[Module Introduction 1](#_Toc51533448)

[1.1 Authorization 1](#_Toc51533449)

[1.2 Adding UserId to Posts 1](#_Toc51533450)

[1.3 Protecting Resources with Authorization 1](#_Toc51533451)

[1.4 Passing the UserId to the frontend 2](#_Toc51533452)

[1.5 Using the UserId on the frontend 2](#_Toc51533453)

# Module Introduction

# 1.1 Authorization

We don’t want every user edit or delete every posts. User should be connected to posts created, in the database so that we want connect which helps to store the id of user. When user tries to delete a post, we want to check the user who did this is the user who created the post.

# 1.2 Adding User ID to Posts

Navigate to our backend code, this is where we can find which user created which posts in database, each posts can be only created by one user.

If UserId of user who sending the request matches the userId of the post.

Navigate to backend/models/posts.js and include another property ‘creator’, this is to tell who has created the post and we are referencing it to User

creator: { type: mongoose.Schema.Types.ObjectId, ref: "User", required: true }

Navigate back to the backend/routes/posts.js, we need to include the creator field while we are creating/saving post.

But if you realize, on our UI when we are creating a post we are not sending in creator id.

In order to fetch this information , we need to navigate back to the login feature, where upon successful login after the checkUserAuth middle aware , we get a jwt token and this jwt token is decoded , which will have the details of user logged in i.e. email Id & userId

Navigate to middleware/check-auth.js and set the decoded email ID & userId into a new object and upon adding it to

req.userData, the request will be passed along and can be accessible in the function where we create/save post.

const decodedToken = jwt.verify(token, "test\_secret\_key");

req.userData = { email: decodedToken.email, userId: decodedToken.userId };

In order to validate, if the req.userData has these new fields,

Navigate to backend/routes/posts.js, include ‘creator’ field in the request of .post function as below

router.post(

"",

checkAuth,

multer({ storage: storage }).single("image"),

(req, res, next) => {

const url = req.protocol + "://" + req.get("host");

const post = new Post({

title: req.body.title,

content: req.body.content,

imagePath: url + "/images/" + req.file.filename,

**creator: req.userData.userId**

});

post.save().then(createdPost => {

res.status(201).json({

message: "Post added successfully",

post: {

...createdPost,

id: createdPost.\_id

}

});

});

}

);

Since now the creator id is saved into database for the user created, we will need to see if we are able to fetch it in our angular code,Navigate to post.service.ts, getPosts function and after the response add log to see if the response has the creator id for the user. After this, include this property into the postData

getPosts(postsPerPage: number, currentPage: number) {

const queryParams = `?pagesize=${postsPerPage}&page=${currentPage}`;

this.http

.get<{ message: string; posts: any; maxPosts: number }>(

"http://localhost:3000/api/posts" + queryParams

)

.pipe(

map(postData => {

return {

posts: postData.posts.map(post => {

return {

title: post.title,

content: post.content,

id: post.\_id,

imagePath: post.imagePath,

creator: post.creator

};

}),

maxPosts: postData.maxPosts

};

})

)

Next step, is to use that id on the backend to deny any update/delete requests made by another user for a given post.

# 1.3 Protecting Resources with Authorization

Navigate to backend code - router, put function. Here we need to ensure only the user created the post can edit it. For that, we can update the .put route and Post.UpdateOne function, to just include another argument, which has creator id, this will ensure only the matching id & userId can be allowed to update the post.

Post.updateOne(

{ \_id: req.params.id, creator: req.userData.userId },

post

)

After this try to login into the app with new userid [test3@test.com](mailto:test3@test.com) , here you can see there is already existing post, because we haven’t added the frontend logic here.

Now try to edit that post with some text change and save it, it will not save because the post was not created by this id [test3@test.com](mailto:test3@test.com) instead was created by previous id. Now login with previous id and try to edit the post, it will work because the post was created by that user.

But if you see, whenever the update fails because even after getting a success response the updated value was not reflected on UI, so it seems we are not throwing a meaningful error to the user.

To check it, let’s add some logs when we get the mongodb response after the update is done and see if we can use any indicator to determine if update failed /succeeded.

Let’s login to [test3@test.com](mailto:test3@test.com), the user who didn’t create the post, because this is the user when tried to edit the post got the request failed. Edit the post and save. Check the logs

**Console.log (result) after the .update One response callback**

Navigate to terminal and you will see the mongodb response which has nModified field which says how many records were modified/updated.

If nModified is 0 it infers the updated failed else it was successful.

Now log back in the previous user who has created the post and validate the logs you will see nModified as 1 which infers the update was successful, we can use this field and add as condition to send success/failure response.

const post = new Post({

\_id: req.body.id,

title: req.body.title,

content: req.body.content,

imagePath: imagePath,

creator: req.userData.userId

});

Post.updateOne(

{ \_id: req.params.id, creator: req.userData.userId },

post

).then(result => {

Console.log(result)

**if (result.nModified > 0) {**

**res.status(200).json({ message: "Update successful!" });**

**} else {**

**res.status(401).json({ message: "Not authorized!" });**

**}**

});

Now log back in with the user who didn’t create the original post which is [test3@test.com](mailto:test3@test.com) and try to edit the post, now it will return an error in the network tab as 401 and the post is not updated. We didn’t yet handle the error or spinner on UI when it returns a 401.

Now let’s add the same logic to the deletion route in backend/routes/posts.js.

1. Is to ensure creator id is passed in the request
2. result.n will determine if the post has been deleted or not.

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

Post.deleteOne({ \_id: req.params.id, creator: req.userData.userId }).then(

result => {

console.log(result);

if (result.n > 0) {

res.status(200).json({ message: "Deletion successful!" });

} else {

res.status(401).json({ message: "Not authorized!" });

}

}

);

Now test it with same userid test3#test.com who didn’t create the post and try to delete it. It will return a 401. Now log back in the valid userid who has created the post and delete the post it will return 200. This completes our server side authorization.

# 1.4 Passing the User Id to the frontend

We need to add similar authorization to our front end code, meaning we should allow only the user who has created the post to edit/delete instead block the other user who tries to edit/delete the posts which he didn’t create.

If you see, the current front end code, just displays all the posts. Let’s work on adding this authorization on frontend.

To do this, we need to go to post list component and check the condition for Edit/delete, those are enabled when user is authenticated but not authorized which means who actually created the post.

We need the userId to the frontend code.

In the backend/routes/user.js, in the /login route where we return the token we need to send userId as well in the response.

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

const token = jwt.sign(

{ email: fetchedUser.email, userId: fetchedUser.\_id },

"test\_secret\_key",

{ expiresIn: "1h" }

);

res.status(200).json({

token: token,

expiresIn: 3600,

userId: fetchedUser.\_id

});

This way we are ensuring whenever user is logged in we are sending the userId as well in the response. This also means we can store this information in the front end in the authservice.ts.

Navigate to Front end code src/app/auth/authService.ts and add a new field UserId.

private userId: string;

in the login function is successful, we need to set this field and also in the typescript post constructor as below which will tell this request will give a response userId field as well.

.post<{ token: string; expiresIn: number, userId: string }>(

this.userId = response.userId;

**Full code:**

login(email: string, password: string) {

const authData: AuthData = { email: email, password: password };

this.http

.post<{ token: string; expiresIn: number, userId: string }>(

"http://localhost:3000/api/user/login",

authData

)

.subscribe(response => {

const token = response.token;

this.token = token;

if (token) {

const expiresInDuration = response.expiresIn;

this.setAuthTimer(expiresInDuration);

this.isAuthenticated = true;

this.userId = response.userId;

this.authStatusListener.next(true);

const now = new Date();

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

console.log(expirationDate);

this.saveAuthData(token, expirationDate, this.userId);

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

}

});

}

This means we can use our UserId in our application, for that we need to create a method and return this userid field, this method can be invoked from different layers in the frontend code.

Navigate to post.list.ts component and add the userId field

userId: string;

and in the ngOnInit fetch the userid from authservice

ngOnInit() {

this.isLoading = true;

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

**this.userId = this.authService.getUserId();**

this.postsSub = this.postsService

the same should be updated under authStatusSub in case we have updated Userid when authentication changes.

this.authStatusSub = this.authService

.getAuthStatusListener()

.subscribe(isAuthenticated => {

this.userIsAuthenticated = isAuthenticated;

this.userId = this.authService.getUserId();

});

In case the user logs out, we need to ensure the userid is reset to null and should be removed from local storage. Navigate to auth.service.ts

**In the logout() function ,set userid to null**

logout() {

this.token = null;

this.isAuthenticated = false;

this.authStatusListener.next(false);

this.userId = null;

clearTimeout(this.tokenTimer);

this.clearAuthData();

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

}

Additional we have got our autoAuthUser, we need to update the userid from authInformation

autoAuthUser() {

const authInformation = this.getAuthData();

if (!authInformation) {

return;

}

const now = new Date();

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

if (expiresIn > 0) {

this.token = authInformation.token;

this.isAuthenticated = true;

**this.userId = authInformation.userId;**

this.setAuthTimer(expiresIn / 1000);

this.authStatusListener.next(true);

}

}

Along with this, since we are storing the authInformation in local storage, we need to store the information as well in local storage.

First in **saveAuthData, and also ensure it is added to the parameter as userId**

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

and in localStorage

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

localStorage.setItem("token", token);

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

**localStorage.setItem("userId", userId);**

}

And in clearAuthData(), we need to delete the userId item

private clearAuthData() {

localStorage.removeItem("token");

localStorage.removeItem("expiration");

**localStorage.removeItem("userId");**

}

Next in getAuthData()

private getAuthData() {

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

const expirationDate = localStorage.getItem("expiration");

**const userId = localStorage.getItem("userId");**

if (!token || !expirationDate) {

return;

}

return {

token: token,

expirationDate: new Date(expirationDate),

**userId: userId**

}

}

Also since we have added a parameter ‘userId in saveAuth Data, we need to provide it under the main function login( … )

this.saveAuthData(token, expirationDate, this.userId);

login(email: string, password: string) {

const authData: AuthData = { email: email, password: password };

this.http

.post<{ token: string; expiresIn: number, userId: string }>(

"http://localhost:3000/api/user/login",

authData

)

.subscribe(response => {

const token = response.token;

this.token = token;

if (token) {

const expiresInDuration = response.expiresIn;

this.setAuthTimer(expiresInDuration);

this.isAuthenticated = true;

this.userId = response.userId;

this.authStatusListener.next(true);

const now = new Date();

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

console.log(expirationDate);

**this.saveAuthData(token, expirationDate, this.userId);**

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

}

});

}

# 1.5 Using the User Id on the frontend

We are handling the userid now, we can use it to authorize the user navigate to the post.list.component.html. under add condition to compare userId with post.creator

<mat-action-row **\*ngIf="userIsAuthenticated && userId === post.creator">**

<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>

We need to ensure the post.model.ts has this new property ‘creator’ and also it should be set.

Add this property in post.model.ts

creator: string;

After this navigate to post.service.ts and ensure the creator field is present

Under **getPosts**() as below

id: post.\_id,

imagePath: post.imagePath,

creator: post.creator

Similarly in **getPost()**

getPost(id: string) {

return this.http.get<{

\_id: string;

title: string;

content: string;

imagePath: string;

creator: string;

}>("http://localhost:3000/api/posts/" + id);

}

**In updatePosts() as**

content: content,

imagePath: image,

creator: null

**Instead of handling this on front end, for updatePosts whenever user does,**

Let’s navigate to backend/routes/posts.js and **into .put function,** we need to include the creator field

const post = new Post({

\_id: req.body.id,

title: req.body.title,

content: req.body.content,

imagePath: imagePath,

creator: req.userData.userId

});

Post.updateOne(

Try to build and see, it will throw an error because our post.create.component doesn’t have this creator field because we do a getPost () in ngOninit

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

this.isLoading = false;

this.post = {

id: postData.\_id,

title: postData.title,

content: postData.content,

imagePath: postData.imagePath,

**creator: postData.creator**

};

Now try to validate different scenarios.

1. Login into with userId as example test3@test.com, which didn’t create the post, expectation is we shouldn’t be seeing the edit/delete option.
2. Login into with userId which created the post, expectation is we should be seeing edit/delete option.

This completes our authorization module where we handled how to protect resources both on server and frontend.