Advanced Web Programming

Lab13

Authentication and authorization in Web App

Current state: After lab 12 you should have implemented CRUD applications to manage the comment list. The Frontend Application consists of 2 main views: a list of all comments and a view for adding a new comment. The front-end application should be fed with data coming from the backend - a self-written web server, and the data should come from the Atlas MongoDB cloud.
Today we will expand our application with authentication and authorization mechanisms. This will allow us to distinguish the users of the application and determine which permissions and possibilities each of them will have. Every web application that handles user-specific data needs to implement authentication. Knowing how to do this is important for Vue developers, and that's why this lab instruction aims to shed the spotlight on.
In this manual:

- You learn what does mean login, registry or logout in Vue App.
- How web server validate you and your permissions in web app
- What kind of data web server send as response to login action and how frontend app works with it.
- You will look at using Vue Router to handle authentication and access control for different parts of your Vue.js application.

Introduction to Authentication

Authentication is a very necessary feature for applications that store user data. It's a process of verifying the identity of users, ensuring that unauthorized users cannot access private data — data belonging to other users. This leads to having restricted routes that can only be accessed by authenticated users. These authenticated users are verified by using their login details (i.e. username/email and password) and assigning them with a token to be used in order to access an application's protected resources.

So authentication simply means the action of proving or showing that something is true or valid. So, we can say that user authentication is the action of validating a user, while authorization is permitting or granting the user access to web resources, features, or views. In the following section, we shall be looking at how a user can be authenticated and authorized using a JSON Web Token.

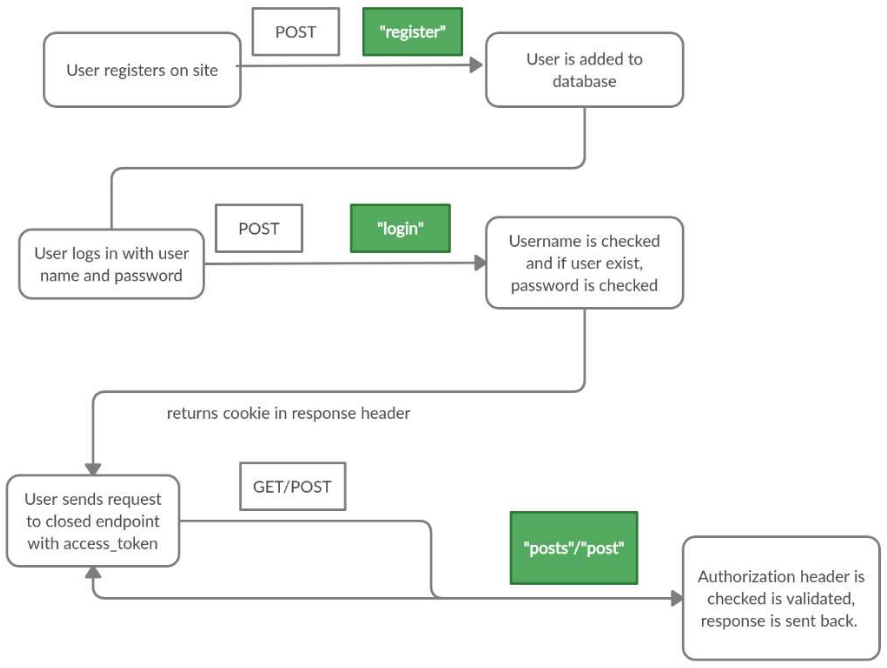
In terms of web applications, we have reached a stage where the application data doesn't only benefit you alone. It is often required to enable a third-party application access/usage of your backend applications and APIs to unleash the full potential of your application. For example, Twitter provides an API to grab its data (for an authenticated user, of course) and makes this usable for all third-party applications. Thus, there's always a reason to have a secure backend application or API.

How authentication works in Web App

This is how authentication works in web app:

- The flow starts from the Vue application where users send the REST API implementing the JWT authentication endpoints,
- the Node/Express auth endpoint generates JWT tokens upon registration or login, and send them back to the Vue application
- the Vue application uses local storage to persist the JWT token,
- the Vue application verifies the JWT tokens when rendering protected views or using some protected action
- the Vue application sends the JWT token back to Node auth server when accessing protected API routes/resources.

Details of the authentication flow are shown in the figure below


User registers on site
POST
"register"
User is added to database
User logs in with user name and password
POST
"login"
Username is checked and if user exist, password is checked
returns cookie in response header
User sends request to closed endpoint with access_token
GET/POST
"posts"/"post"
Authorization header is checked is validated, response is sent back.

After successfully logging in a user, the access token alongside some data will be received in the Vue app, which will be used in setting the cookie and attached in the request header to be used for future requests. The backend will check the request header each time a request is made to a restricted endpoint. Today modern web app using JWT Token as a access token format.

JSON Web Token

JWT, an acronym for JSON Web Token, is an open standard that allows developers to verify the authenticity of pieces of information called claims via a signature. This signature can either be a secret or a public/private key pair. Together with the header and the payload, they can be used to generate or construct a JWT, as we will get to see later.

JWTs are commonly used for authentication or to safely transmit information across different parties. As I said before common flow for JWT-based authentication systems: once a user has logged into an app, a JWT is created on the server and returned back to the calling client.

Each subsequent request will include the JWT as an authorization header, allowing access to protected routes and resources. Also, once the backend server verifies the signature is valid, it extracts the user data from the token as required. Note that in order to ensure a JWT is valid, only the party holding the keys or secret is responsible for signing the information.

In this lab, we will be focusing on using JWT to perform authentication requests on a Vue.js client app with a Node.js/ExpressJS backend. But first, let's review how JWT works in a nutshell.

How JWT authentication works

In JWT authentication-based systems, when a user successfully logs in using their credentials, a JSON Web Token will be returned back to the calling client. The Server generated a JWT from user login data and send it to the Client. The Client saves the JWT and from now, every Request from Client should be attached that JWT (commonly at header). The Server will validate the JWT and return the Response. Whenever the user wants to access a protected route or resource, the user agent sends the same JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like this:

```
Authorization: Bearer <token>
```

For a user to be granted access to a protected resource, the server routes will have to check for the presence of a valid JWT in the Authorization header. As a bonus, sending JWTs in the Authorization header also solves some issues related to CORS. This applies even if the app is served from an entirely different domain.

Important!!!. Even if JWTs are signed, the information is still exposed to users or other parties because the data are unencrypted. Therefore, users are encouraged not to include sensitive information like credentials within a JWT payload. Additionally, tokens should always have an expiry.

the Server generated a JWT from user login data and send it to the Client. The Client saves the JWT and from now, every Request from Client should be attached that JWT (commonly at header). The Server will validate the JWT and return the Response.

In this point lets go to implement authentication and authorization in Web app.
We start implemented authentication from Backend. From the docs, you'll notice few endpoints are attached with a lock. This is a way to show that only authorized users can send requests to those endpoints. The unrestricted endpoints are the /register and /login endpoints. An error with the status code 401 should be returned when an unauthenticated user attempts to access a restricted endpoint.

Step 1. Securing Node and Express RESTful API with Json Web Token (JWT)

What steps should be taken on the backend side:

- Updating our web server folder structure
- Creating Schema and Model for User
- Defining User Authentication Endpoints and Routes
- Creating Controller functions for User Authentication and Signing JWT Token
- Applying User Authentication Controller on our app endpoints
- Verifying JWT Token
- Testing API on Postman

Exercise 1. Updating Folder Structure

Firstly, we need to update the file structure of the existing in our backend directory. We define and add some new files in our backend directory:

```
app/controllers/authController.js
app/models/userModel.js
```

Let's run the snipet below in the command line to install other dependences to be used.

```
npm install jsonwebtoken  --save

npm install bcryptjs  --save
```

Exercise 2. Creating Schema and Model for User

Just as we created schema and model for the Comment application API. We are also going to do the same here to create a schema for the user. To get started with this, let's open *userModel.js* file in the model folder and follow the steps as shown in the snippet below.

```
    'use strict';

    // Import mongoose
    const  mongoose = require("mongoose");

    // Import bcryptjs - for password hashing
    const  bcrypt = require('bcryptjs');

    // Declare schema and assign Schema class
    const  Schema = mongoose.Schema;

    // Create Schema Instance for User and add properties
    const  UserSchema = new  Schema({
```

```
    fullName: {
      type:  String,
      trim:  true,
      required:  true
    },

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

    hash_password: {
      type:String,
      required:true
    },

    role: {
      type:String,
      required:true
    },

    createdOn: {
      type:  Date,
      default:  Date.now
    }

    });

    //Create a Schema method to compare password
    UserSchema.methods.comparePassword = function(password){
         return  bcrypt.compareSync(password, this.hash_password);
    }

    // Create and export User model
    module.exports = mongoose.model("User", UserSchema);
```

From the above snippet, we imported bcryptjs, which has already been installed. We used bcrypt.compareSync function in the snippet above takes only 2 arguments and returns a boolean value true or false.
It is important to know that bcryptjs is a password hashing function that does not only incorporating a salt "Salt (cryptography)") to protect against rainbow table attacks, also provide resistant to brute-force search attacks even with increasing computation power. bcryptjs and bjcrypt work as the same but the former is an optimized bcrypt in JavaScript with zero dependencies.

Exercise 3. Defining User Authentication Endpoints and Routes

The endpoints needed for the user authentication (to register and login users) are shown in the table below. Update routes file defining new endpoints.

```
/auth/register
|Method | Description |
|--|--|
|POST| Create user |

/auth/login
|Method| Description |
|--|--|
|POST | Login and authenticate user |
```

Now that we have defined user authentication endpoints, the next is to create and define controller functions for those endpoints and updates the routes

Exercise 4. Creating Controller functions for User Authentication and Signing JWT Token
Before authentication can take place, user credentials must have been submitted, In the part, we are going to write controller functions to register user credentials and to authenticate the user before access to restricted resources.
There are 2 main functions for Authentication:
- register: create new User in database (role is user if not specifying role)
- signIn:
  - find username of the request in database, if it exists
  - compare password with password in database using bcrypt, if it is correct
  - generate a token using jsonwebtoken
  - return user information & access Token

Open the *authController.js* file and follow comment guide in the snippet below:

```
    // import User model
    const User = require("../models/userModel");

    // import jsonwebtoken
    const jwt = require('jsonwebtoken'),

    // import bcryptjs - hashing function
    const  bcrypt = require('bcryptjs');

    //DEFINE CONTROLLER FUNCTIONS

    // User Register function
    exports.register = (req, res) => {
       let newUser = new User(req.body);
         newUser.hash_password =   bcrypt.hashSync(req.body.password, 10);
         newUser.role = “user”;

         newUser.save((err, user) => {
              if (err) {
                  res.status(500).send({ message: err });
              }
              user.hash_password = undefined;
              res.status(201).json(user);
         });
    };

    // User Sign function
    exports.signIn = (req, res) => {
        User.findOne({
            email: req.body.email
        }, (err, user) => {
            if (err) throw err;
            if (!user) {
               res.status(401).json({ message: 'Authentication failed. User not found.' });
            } else if (user) {
               if (!user.comparePassword(req.body.password)) {
                 res.status(401).json({ message: 'Authentication failed. Wrong password.' });
               } else {
                res.json({ token: jwt.sign({ email: user.email, fullName: user.fullName, _id:
user._id , role: user.role}, config.secret, { expiresIn: 86400 })
```

```
                });
            }
        }
    });
};

// User Register function
exports.loginRequired = (req, res, next) => {
  if (req.user) {
      res.json({ message: 'Authorized User, Action Successful!'});
  } else {
      res.status(401).json({ message: 'Unauthorized user!' });
  }
};
```
From the above snippet, we imported User model, bcryptjs and jsonwebtoken libraries. The bcrypt.hashSync function used in the register controller function takes plain text password and rounds(in number) uses a random segment (salt) to generate the hash associated with the password. This was stored along with the password.
You can Recall from the user model that the compareSync method recalculates the hash of the password entered by the user and compares with the one entered when registering and see if they match as returned by the comparePassword function.
Also, from the Sign In controller function, the jsonwebtoken sign method jwt.sign({…}, 'secret') is used to sign web token.

Format sign method:
```
      jwt.sign(payload, privateKEY, options)
```

Then, privateKey we define in `config.js` in the same directory and add the following to it.
Jsonwebtoken functions such as verify() or sign() use algorithm that needs a secret key (as String) to encode and decode token. In the app/config folder, create auth.config.js file with following code:

auth.config.js
```
module.exports = {
  'secret': 'your_secret'
};
```
You can create your own secret String

In this example when you use registry function you only create new account in MongoDB. After registry you should login in frontend app to work as a authenticated user. If you want the registration to log user automatically you should implement registry function for example like this (bellow).

Let's define the route for registering and login a new user:
```
// User Register function
 exports.register = (req, res) => {
   let newUser = new User(req.body);
   newUser.hash_password =   bcrypt.hashSync(req.body.password, 10);
   newUser.save((err, user) => {
              if (err) {
                  res.status(500).send("There was a problem registering the user.");
              }
              let token = jwt.sign(
                  { email: user.email, fullName: user.fullName, _id: user._id },
                    config.secret,
                  { expiresIn: 86400 } // expires in 24 hours
              );
             res.status(200).send({
                 auth: true,
                 token: token,
                 user: user
           });
   });
};
```

First, you pass the request body [req.body.fullName, req.body.email, req.body.password] to a database method (which you will define later), and pass a callback function to handle the response from the database operation. As expected, you have defined error checks to ensure we provide accurate information to users.
When a user is successfully registered, you select the user data by email and create an authentication token for the user with the jwt package you imported earlier. You use a secret key in your config file to sign the auth credentials. This way, you can verify a token sent to your server and a user cannot fake an identity.

```
// Login
router.post('/login', (req, res) => {
    User.findOne({ (req.body.email, (err, user) => {
        if (err) return res.status(500).send('Error on the server.');
    if (!user) return res.status(404).send('No user found.');
    let passwordIsValid = bcrypt.compareSync(req.body.password, user. hash_password);
    if (!passwordIsValid) return res.status(401).send({
      auth: false,
      token: null
    });
    let token = jwt.sign(
      { email: user.email, fullName: user.fullName, _id: user._id },
      config.secret,
      { expiresIn: 86400 } // expires in 24 hours
    );
    res.status(200).send({
      auth: true,
      token: token,
      user: user
    });
  });
});
```

For login, you will use bcrypt to compare your hashed password with the user-supplied password. If they are the same, you log the user in. If not, feel free to respond to the user how you please.

Exercise 5. Applying User Authentication Controller on Comment API endpoints
Open commentRoutes.js file and updates the HTTP request(POST, PUT, DELETE) that modifies resources with authentication handler as shown in the snippet below:

```
'use strict';

    module.exports = function(app) {
        let  commentList = require('../controllers/commentController');
        let  userHandlers = require('../controllers/authController');

    // commentList Routes

    // get and post request for /comments endpoints
        app
        .route("/comments")
        .get(commentsList.listAllComments)
       .post(userHandlers.loginRequired, commentsList.createNewComment); // update with login
handler

    ……………..   ///  rest of request defined in Routes politics

    // post request for user registration
        app
        .route("/auth/register")
        .post(userHandlers.register);

    // post request for user log in
        app
        .route("/auth/sign_in")
        .post(userHandlers.signIn);
        };
```

Of course I give you only some example how to define endpoints. Rest of endpoints define/update by own.   As you see I implement a authorization if I have grants to add new Comment in our app   ->   .post(userHandlers.loginRequired, commentsList.createNewComment);

Exercise 6. Verifying JWT Token
Recall that JWT token were created in the process of signing in or in the second version of implementation in registry too. Therefore, in other to grant access to users on authenticated

endpoints, the token initailly created my be verified using JWT verify method. Open the server file, *server.js* and updates with Token verification.

```
        'use strict'


        // import jsonwebtoken
            const  jwt  =  require('jsonwebtoken');

        // Token Verification
            app.use((req, res, next) => {
                if (req.headers && req.headers.authorization &&
req.headers.authorization.split(' ')[0] === 'JWT') {
                jwt.verify(req.headers.authorization.split(' ')[1], config.secret, (err,
decode) => {
                if (err) req.user = undefined;
                req.user = decode;
                next();
                    });
                } else {
            req.user = undefined;
            next();
                }
            });

        // API endpoint
```

Exercise 7. Test API on Postman

Test Authenticated endpoints using token in authorization in headers or not.

Step 2. Authentication in Vue

Next step is implementation authentication in Vue app.

Exercise 8. In Vue app create two view: login and registry. To sign up new user should get name, email and password. For all users you should define always "user" role. Admin will be define directly in database, we don't plan registry admin account by app. In login view user give only email and password. Implement login logic. You should save in global object information about your status – you are authenticate, your name (using when you create new Comment – you don't have to write it manually and role assigned to your account.

Login Action

Here is where the main authentication happens. When a user fills in their username and password, it is passed to a User which is a FormData object, the LogIn function takes the User object and makes a POST request to the /login endpoint to log in the user. Remember that then you login response form server has JWT token. You should read information about user from JWT and next save JWT token in localstorage.

Comparing with Session-based Authentication that need to store Session on Cookie, the big advantage of Token-based Authentication is that we store the JSON Web Token (JWT) on Client side using Local Storage for Browser.

To read token we should decoded JWT. You can use for example this ->

```
const verifyJWT = (payload) => {
        return jwt.verify(payload, publicKEY, verifyOptions)
}
```

Note: JWT has a verify method that synchronously verifies a given token, using a secret or a public key and options for the verification. Once the token is verified, a decoded value of that token is returned.

Or better solution is this ->

```
import jwt_decode from 'jwt-decode';

var token = 'eyJ0eXAiO.../// example jwt token';

var decoded = jwt_decode(token);
console.log(decoded);
/*{exp: 10012016 name: john doe, role:['user']}*/
```

Most importantly, to use jwt_decode we have installed vue-jwt-decode.

Very simple code from Login component may looks like this:

```
loginUser() {
      try {
        //  send email and password to server
        let token = response.data.token;
        localStorage.setItem("user", token);
        // navigate to a protected resource
        this.$router.push("/home");
      } catch (err) {
        console.log(err.response);
      }
    }
  }
```

As we can see above, once we log in to our app via the backend API, we are setting the returned token to localStorage. Once that happens, we are navigating to the Home component using the router push API.

Here, as a means of demonstrating the entire flow, we retrieve the token from local storage, decode said token using the vue-jwt-decode package, then finally append that to the user object, which we can then display in our Home component:

```
getUserDetails() {
      // get token from localstorage
      let token = localStorage.getItem("user");
      try {
      //decode token here and attach to the user object
      let decoded = VueJwtDecode.decode(token);
      this.user = decoded;
```

```
        } catch (error) {
          // return error in production env
          console.log(error, 'error from decoding token')
        }
      },

 logUserOut() {
        localStorage.removeItem("user");
        this.$router.push("/login");
      }
```

In our app we should find a functions with functionalities to get the state. It can be used in multiple components to get the current state. The isAuthenticatated function checks if the state.user is defined or null and returns true or false respectively. StateUser return state.user respectively value.

Exercise 9. Define new component NavBar.vue. It should be the component for our navigation bar, it links to different views of our component been routed here. Each router link points to a route/page on our app.
The v-if="isLoggedIn" is a condition to display the Logout link if a user is logged in and hide the Register and Login routes. We have a logout method which can only be accessible to signed-in users, this will get called when the Logout link is clicked. It will dispatch the LogOut action and then direct the user to the login page.

Protecting Vue Routes with Navigation Guards

Now we have implemented authentication. We know who the user are. Sometimes we have to protect some resource, action or views by some of the users. For example in our app, a user who is not logged in is called a guest. The guest should see only list of comments. He shouldn't add new comments or hide/show comments or delete or edit any of comments. So guests shouldn't go to AddNewComment view for example.

How to do it? Solution is protecting view by using navigation guards.

Navigation guards are a specific feature within Vue Router that provide additional functionality pertaining to how routes get resolved. They are primarily used to handle error states and navigate a user seamlessly without abruptly interrupting their workflow.

There are three main categories of guards in Vue Router: Global Guards, Per Route Guards and In Component Guards. As the names suggest, **Global Guards** are called when any navigation is triggered (i.e. when URLs change), **Per Route Guards** are called when the associated route is called (i.e. when a URL matches a specific route), and **Component Guards** are called when a component in a route is created, updated or destroyed. Within each category, there are additional methods that gives you more fine grained control of application routes. Here's a quick break down of all available methods within each type of navigation guard in Vue Router.

Global Guards

- **beforeEach**: action before entering any route (no access to this scope)
- **beforeResolve**: action before the navigation is confirmed, but after in-component guards (same as beforeEach with this scope access)
- **afterEach**: action after the route resolves (cannot affect navigation)

Per Route Guards

- **beforeEnter**: action before entering a specific route (unlike global guards, this has access to this)

Component Guards

- **beforeRouteEnter**: action before navigation is confirmed, and before component creation (no access to this)
- **beforeRouteUpdate**: action after a new route has been called that uses the same component
- **beforeRouteLeave**: action before leaving a route

Protecting Routes

To start, know that there are multiple navigation guard functions available to us. In this case, we will use beforeEach which fires every time a user navigates from one page to another and resolves before the page is rendered.
We link the function up to our router. We pass three arguments to the function. The route they're attempting to go to, the route they came from and next.

```
router.beforeEach((to, from, next) => {


})
```

The method takes three parameters — `to`, `from`, and `next`. `to` is where the user wishes to go, `from` is where the user is coming from, and `next` is a callback function that continues the processing of the user request. Your check is on the `to` object.

You will check:

- if route `requiresAuth`, check for a `jwt` token showing the user is logged in.
- if route `requiresAuth` and is only for admin users, check for auth and check if the user is an admin
- if route requires `guest`, check if the user is logged in

You will redirect the user based on what we are checking for. You will use the `name` of the route to redirect, so check to be sure you are using this for your application.

Vue Router allows you to define a meta on your routes so you can specify additional behavior.

In your case, you have defined:

- some routes as guest: which means only users not authenticated will see it
- some to require authentication [requiresAuth]: which means only authenticated users will see it
- and the last one to be only accessible to admin users [is_admin]

```
// Meta Handling
router.beforeEach((to, from, next) => {
  if (to.matched.some(record => record.meta.requiresAuth)) {
    if (localStorage.getItem('jwt') == null) {
      next({
        path: '/login',
        params: { nextUrl: to.fullPath }
      })
    } else {
      let user = JSON.parse(localStorage.getItem('user'))
      if (to.matched.some(record => record.meta.is_admin)) {
        if (user.is_admin == 1) {
          next()
        } else {
          next({ name: 'userboard' })
        }
      } else {
        next()
      }
    }
  } else if (to.matched.some(record => record.meta.guest)) {
    if (localStorage.getItem('jwt') == null) {
      next()
    } else {
      next({ name: 'userboard' })
    }
  } else {
    next()
  }
})
```

```
export default router
```

Next

next is actually a function and it's very interesting. next has to be called in order to resolve our guard. So every logic path needs to hit next in some way.
There are multiple ways to call next, but I want to point out three.
next() sends you to the next set of logic. If there isn't any, the navigation is confirmed and the user gets sent to to.
next(false) this sends the user back to from and aborts their attempted navigation.
next(<route>) this sends the user elsewhere, wherever you determine that is.
We're going to make use of the first and last options in our navigation guard.

Ok, so now we need to determine in what circumstances we're sending the user one place or the next. In our case, we want to check for authenticated users. However, not all of our pages require you to be authenticated. We can define that in our route metadata so we know if we care about checking or not.

```
const routes = [

  {

    path: '/',

    component: Home,

    meta: {

      requiresAuth: false,

    },

  }

]
```

That means that the first thing we want to look at is whether our to route requiresAuth.
If it does, we have more to write. If it doesn't, we've decided the user can navigate there, so we'll call next(). In this case, nothing follows that call, so next() will confirm the navigation.

```
router.beforeEach((to, from, next) => {
  if (to.matched.some((record) => record.meta.requiresAuth)) {

  } else {

     next()

  }

})
```

Now we're adding the last piece of the puzzle. If requiresAuth is true, then we want to check if our user is authenticated.
Note that we're not showing the implementation of isAuthenticated. This can be any number of things. We're just making the assumption we have a way to check.
If our user is authenticated, we want to confirm the navigation. Otherwise, we'll send them to the login page.

```
router.beforeEach((to, from, next) => {

  if (to.matched.some((record) => record.meta.requiresAuth)) {

    if (isAuthenticated()) {

      next()

    } else {

      next('/login')

    }

  } else {

    next()

  }

})
```

}

To be honest, the implementation below is a bit cleaner. No need to call next() twice, less if/else logic. But for some reason I've never liked checking on a false case, it just seems a bit confusing. However, others may feel differently, so know that this is also an option.

```
router.beforeEach((to, from, next) => {

  if (to.matched.some((record) => record.meta.requiresAuth)) {

    if (!isAuthenticated()) {

       next('/login')

    }

  } else {

     next()

  }

})
```

Initially, I had code that looked like this. And it works just the same! But I couldn't figure out the return piece of the puzzle and why I needed it. So I wanted to explain.

```
router.beforeEach((to, from, next) => {

  if (to.matched.some((record) => record.meta.requiresAuth)) {

    if (isAuthenticated()) {

      next()

      return

    }

    next('/login')

  }
  next()

})
```

You could also write return next(), whichever you prefer.
next() only confirms the navigation if there are no hooks left in the pipeline. Since there were no else statements, only fall through behavior, next() didn't confirm anything, it just sent you to the "next" thing.
That didn't matter for records that didn't require auth, because you were sent to the final next() which was the end of the road. But for authenticated users, they'd always end up on the login page. So in order to make it work, the return is needed. It prevents the code that follows from being executed and confirms the navigation.

Conclusion

We've built a navigation guard to check authentication for all our pages. Navigation guards, and vue-router in general, are incredibly powerful.

Exercise 10. Implement protection from possibility to use addNewComponent by guest in our app. Lets app redirect guest to login view.

Exercise 11. Define new view in our app. It will be Admin view. Only admin (user with that role) to be able to goes to that view. Another users (logged or guests) don't be able to use this view.

Exercise 12. Let user who is not the owner of the item in commentList cannot perform an action (delete, update) or change its status (hide or show). Only owner or admin should be able to do it.