T7 - MSc Pool

T-POO-700

Authentication

Bootstrap

0.7

The objective of this bootstrap is to set up a to-do list that takes into account authenticated users with different roles and permissions, as well as a system of associated skills.

Do not hesitate to go back to your previous API-bootstrap-theme to update it !

Step 1 : Setting up authentication

Part 1 : API

Let's start from the API of your todo list.
To allow users to authenticate, you need to handle passwords.

The passwords must be hashed for security reasons. Check Elixir's Bcrypt library.

The authentication is based upon **JSON Web Token**, handled ***Joken*** library.
You must first generate a 50-character c-xsrf-token, and then create your JWT containing:

- the c-xsrf-token
- the user id
- the user role
- 30 days expiry time

You will send the JWT in HTTP Only, and return the xsrf-token when connecting.
Whenever you use a route, check the match of the xsrf-token of the request with the one of the JWT.
Using this method, you will also retrieve the user's id and the role of the user to check access to the routes.

Once **your schemas**, your **json views**, your **JWT** and your **endpoints** are modified, you need to create new endpoints.

- `/users /sign_in` (POST)
  to authenticate a user to your platform. It takes in parameter **an email** and **a password**, and returns an xsrf-token, the role of the user and the id of the user
- `/users/sign_up` (POST)
  to register a user. It requires all possible information from a user *(Name, Surname, Email, Password)*
- `/users/sign_out`
  to disconnect a user. It deletes or invalidates the current JWT of the user.

Remember to return the correct HTTP codes!

PART 2 : FRONT-END

Create a new Vue.JS project called *bs-auth*, using *view-cli*, and install the following npm dependencies :

- axios
- vue-router

Create an *Authentication* component connected to the `/sign_in` route using *view-router*.

Your router will be of **vital** importance to you to verify the access rights of a user to a page. It is recommended to create a `'/src/router.js'` file containing all the code related to the router. It can then be imported into the file `'/src/main.js'`.

The template of this component will be a form with the following inputs :

- E-mail
- password

A button, linked to the `'signIn()'` method of your component, will validate the form.
You will store in the Web Storage the data returned by the API : xsrfToken, userId, roleId.

Passwords are **never** sent in plain text, npm packages can help with hashing.

XSRF and JSON Web are **different** tokens! One is stored in an "HttpOnly" cookie, **inaccessible by our front-end**, and automatically transmitted in each request.

The *"localStorage"* stores **strings of characters**. The *"localStorage"* is often favored in *"Store Vue.JS"*, deleted during *"refresh"*

You **must** pass your XSRF token in the `'x-xsrf-token'` header of **ALL** your queries. You now have a component to connect a user. You will need to modify your router to restrict access based on the logged in user, using the `beforeEach()` method, before each page change.

Without the `next()` function, no redirection will be made!

You will create the user registration form, with the following specifications :

- your component will be named *"Registration"*
- it will be connected to the `'/sign_up'` route (of course, this route will be public)

- the *'submit'* button on the form will call the `'signUp()'` method of your component
- the `'signUp()'` method will make a request to the API on the endpoint `'/users/sign_up'` (see part 1)
- if registration has been successful, you will redirect your user to your site's homepage (route `'/'`)

The bootstrap 'Web interfaces' can be useful.

You will finally be able to manage the disconnection of a user, by modifying the component *"App"* and by adding a *"Logout"* button, displayed only if the user is connected.

Using the *"v-on: click"* directive, link your button to a `'disconnect()'`function, which should:

- make a request to the API to disconnect the server-side user
- delete data from *"localStorage"*

Step 2 : Skills Management

Part 1 : API

To identify skills, we have a `label` (its name) and some associations with other schemes :

- `many_to_one` between tasks and skills. Each task can have only one skill.
- `many_to_many` between users and skills.

Each user has several skills.

Set up a CRUD so you can create, modify and delete your skills.

In addition to CRUD, we want to be able to add skills to a user or task.
To do this, you must implement the following routes :

- `/users/:userid/skills/:skillid` (POST)
  to add a skill to a user.
  `userid` is the id that corresponds to the user.
  `skillid` corresponds to the id of the skill you want to add.

- `/users/:userid/skills/:skillid` (DELETE)
  to remove a skill from a user.
  `userid` is the id that corresponds to the user.
  `skillid` is the id of the skill you wish to remove.

- `/tasks/:taskid/skills/:skillid` (PUT)
  to assign a skill to a task.
  `taskid` is the id that corresponds to the task.
  `skillid` is the id of the skill you want to add.

PART 2 : FRONT-END

As for the second part of step 1, the goal is to create the user interface, via a *"SkillsManager"* component, connected to the following routes :

- `'/skills'` to display a list of all the skills and a form for creating a skill
- `'/skills/:skillId'` to display the details of a skill as well as a possibility to modify/delete this skill

Create a *AccountManager* component, connected to the route `'/profile'` and accessible only by a connected user.
It will display and allow to modify the following information:

- last name
- first name
- e-mail
- password (**it will not be displayed**)
- role (**unchangeable**)
- skills

In addition, create a *Users* component, connected to the `'/users'` path, which displays the list of users.

Finally, create the *Tasks* component to manage the tasks. It will be connected to the following routes :

- `'/tasks'` to show the list of all tasks as well as a creation form. Buttons must be present in order to be able to assign to a task, withdraw from it, and change its status.
- `'/tasks/:taskId'` to display a form for editing an existing task. Possible to edit all task fields, associated skills and assigned users. A button will also be present to delete this task.

Once the changes are effective in your router, all you have to do is restrict certain features on the `/tasks` route.

Step 3 : Creating permissions

Part 1 : API

This part will give sense of authentication.
To manage the different permissions, you will need to set up roles. To begin, create a **Role** schema :

- The name of the schema is: `Roles`
- The information we have about these :
  - A **label** that defines the name of the latter.
- But also associations :
  - `one_to_many` with users. Each user has a role.
- But also mandatory values :
  - `Manager`
  - `User` which is the default value of all users.

In order for your roles to remain relevant and your permissions to follow, you must `populate` your database with both roles each time you create your table.

Since roles are fixed elements of your application, there must be no CRUD accessible from your API, except for reading them.

Now let's talk about the rights granted to each of the roles.

- **The Managers**
  - They can create, edit, read and delete all tasks. Which also includes :
    - Add an employee to it.
    - To take it out.
    - Add a skill and remove it from the task.
  - They can add skills to the database, but also edit, read and delete them.
- **The employees**
  - They can assign to a task or withdraw from it.
  - They can change the status of the task.

Employees can assign themselves, or be assigned to a task only if they have the necessary skill (s) for it.

To carry out the actions mentioned above, the following routes will have to be put in place:

- **Manage tasks and associated users :**
  - `/tasks/:taskid/user/:userid`
    - **POST**
    - `taskid` is the id that corresponds to the task.
    - `userid` is the id of the user you want to add.
    - This route is used to assign a user to a task.
  - `/tasks/:taskid/user/:userid`
    - **DELETE**
    - `taskid` is the id that corresponds to the task.
    - `userid` is the id of the user you wish to remove.
    - This route allows you to remove a user from a task.
  - `/tasks/:taskid/status`
    - **PUT**
    - `taskid` is the id that corresponds to the task.
    - This route allows to switch between the states of the task (False / True)

Warning, only the new routes are explained above, remember to change the permissions of the old roads (as for the creation of skills!)

Part 2: Front-End

Last part of this bootstrap: manage the rights of our users on our user interface, using the `beforeEach()` method of class *"VueRouter"*.
These kinds of methods are called *"Hooks"*, or *"Guards"* (more specific term for Vue.JS).

This method is called at each change of course and we will use it to check the role of the user before displaying the page.

We will describe three types of routes :

- **public** : Accessible by any user of the site, connected or not
- **private** : Accessible only by a logged in user
- **administration** : Accessible only by a logged in user who has the *"Manager"* role

Here is a list of all the routes in your application and their type :

- `'/'` : **public**
- `'/sign_in'` : **public**
- `'/sign_up'` : **public**
- `'/skills'` : **administration**
- `'/skills/:skillId'` : **administration**
- `'/profile'` : **private**

- `'/users'` : **administration**
- `'/tasks'` : **private**
- `'/tasks/:taskId'` : **administration**
- View the list of tasks **who has the skills**
- Assign yourself to a task
- To withdraw from a task
- Change the status of a task

While a *"Manager"* will have a list of all tasks regardless of their skills, a creation form and finally a modification link for each task.

Do not use *"v-show"* directive when you want to restrict access to a feature on your site!